VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Technologie Hadoop a její využití při analýze dat

# Hadoop technology and data analysis

2015

Bc. Martin Popelář

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

# Zadání diplomové práce

Student: **Bc. Martin Popelář**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Technologie Hadoop a její využití při analýze dat
Hadoop Technology and Data Analysis

Zásady pro vypracování:

V rámci diplomové práce student prostuduje a popíše framework Hadoop, který obsahuje sadu open-source softwarových komponent určených pro zpracování velkého množství nestrukturovaných a distribuovaných dat. Dále se bude zabývat implementací tohoto frameworku na HPC cluster. V případě neúspěšné implementace přímo na cluster provede implementaci frameworku na virtuální infrastrukturu. Následně student využije Hadoop a jeho rozšíření pro shlukování dat.

Jednotlivé body práce jsou:
1. Prostudovat a popsat technologii Hadoop.
2. Prostudovat a popsat rozšíření pro Hadoop v oblasti analýzu velkých dat.
3. Analyzovat možnosti nasazení Hadoop na HPC cluster.
4. Nasadit Hadoop na HPC cluster Anselm nebo virtuální infrastrukturu.
5. Provést experimenty z oblasti shlukování dat s využitím Hadoop.
6. Vyhodnocení experimentů.

Seznam doporučené odborné literatury:

[1] Chuck Lam: Hadoop in Action, Manning Publications; 1 edition, 2010
[2] Philip Russom: Big Data Analytics, TDWI RESEARCH, 2011

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 01.09.2014
Datum odevzdání: 07.05.2015


doc. Dr. Ing. Eduard Sojka
*vedoucí katedry*

prof. RNDr. Václav Snášel, CSc.
*děkan fakulty*

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2015 .....

Rád bych na tomto místě poděkoval svému vedoucímu diplomové práce Ing. Janu Martinovičovi, Ph.D za cenné rady a národnímu superpočítačovému centru IT4Innovations za poskytnutí přístupu na superpočítač Anselm.

## Abstrakt

Tato práce se zabývá technologií Hadoop a problematikou Big Data. Jsou zde popsány vybrané frameworky z Hadoop ekosystému (Oozie, Hive a HUE) a nasazení Hadoopu v HPC, což bylo hlavním cílem práce. Provedené výkonnostní testy napovídají, jak Hadoop dokáže v HPC prostředí pracovat s přiřazenými zdroji.

**Klíčová slova:** Hadoop, big data, Oozie, Hive, HUE, Mahout, HPC, myHadoop

## Abstract

This thesis is about Hadoop technology and Big Data issue. It includes description of choosen frameworks from Hadoop ecosystem (Oozie, Hive and HUE) and a deployment of Hadoop in HPC environment, which was main goal of thesis. Performance tests suggest how Hadoop works with assigned resources.

**Keywords:** Hadoop, big data, Oozie, Hive, HUE, Mahout, HPC, myHadoop

## Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| open-source | – | program s volně dostupným zdrojovým kódem |
| HW | – | hardware |
| SW | – | software |
| PBS | – | portable bash system |
| SGE | – | sun grid engine |
| HPC | – | high-performance computing |
| URI | – | uniform resource identifier |
| JVM | – | java virtual machine |
| XML | – | extensible markup language |
| HTTP | – | hypertext transfer protocol |
| FTP | – | file transfer protocol |
| SSH | – | secure shell |
| SQL | – | structured query language |
| API | – | application programming interface |
| CSV | – | comma-separated values |
| JSON | – | JavaScript object notation |
| SŘBD | – | systém řízení báze dat |
| RAM | – | random access memory |
| RSA | – | Rivest, Shamir, Adleman |
| ACID | – | Atomicity, Consistency, Isolation, Durability |

# Obsah

# Seznam tabulek

# Seznam obrázků

# Seznam výpisů zdrojového kódu

# 1 Úvod

Žijeme ve světě informací. Informace jsou všude kolem nás a každý se je snaží shromažďovat a následně analyzovat. Velké společnosti jako například Facebook, Google a Twitter generují denně až petabyty dat[2]. Voláme a používáme „chytré telefony", využíváme bankovní služby, internetové či mobilní bankovnictví, platíme kartou, nakupujeme v obchodě či e-shopu, píšeme e-maily, používáme sociální sítě, pořizujeme fotky a videa atd. Což jsou pouze data, které vytvoří člověk sám. Existuje spousta dalších informací, které se o nás sbírají automaticky - webové aplikace zaznamenávají aktivity svých uživatelů tak podrobně, že dokáží říct, kdo a kdy na co kliknul. Systémy si vedou své logy o chybách a dalších aktivitách. Pracovat s těmito daty je běžným způsobem nemyslitelné, proto vzniklo nové odvětví - *Big Data*. Big Data se definují jako data, které splňují tzv. 3V[10]:

- Volume - Příliš velké množství dat
- Velocity - Velká rychlost vznikajících dat a potřebných analýz
- Variety – Různé a proměnlivé formáty dat, strukturovaná i nestrukturovaná data

Objevují se i další „V" jako například Variability (rozmanitost) nebo Veracity (pravdivost). Přesná definice ale není důležitá. Pro nás jsou Big Data označení skutečnosti, že neustále přibývá různých zařízení a systémů, kde data vznikají, neustále roste množství dat, a tím se zvyšují nároky na zpracování dat. Pokud se ovšem podaří tato data zpracovat a správně využít, získají podniky náskok před konkurencí v podobě zpříjemnění služeb pro koncového zákazníka. Společnosti jako Netflix nebo Pandora dokáží svým uživatelům s vysokou pravděpodobností navrhovat filmy či hudbu, které by se jim mohly líbit. Toho však lze dosáhnout jen díky správné analýze dat. Data se dají použít na generování dalších dat, můžeme svoje data efektivně třídit a následně v nich lépe vyhledávat, využití dat je téměř neomezené. Co je však omezené, je náš výpočetní výkon. Dříve, když jsme chtěli Big Data analyzovat, potřebovali jsme počítače s obrovskou výpočetní silou, tzv. superpočítače. Tomu se snaží vyhnout nová technologie *Hadoop*, díky které můžeme toto velké množství dat analyzovat i na běžně dostupných strojích.

V této práci se ale pokusíme Hadoop otestovat i na superpočítači. Prvním cílem práce je spustit Hadoop na superpočítači, což jak už vyplynulo z předchozího textu, není jeho přirozené prostředí. Ke spuštění využijeme připravených skriptů *myHadoop*. V další části práce budeme Hadoop testovat. Vyzkoušíme jeho škálovatelnost, kdy budeme stejnou úlohu spouštět na různě velkých HPC[1] clusterech a porovnávat výsledné časy výpočtu, ale taky využití dostupných zdrojů. Z toho plyne druhý cíl práce - najít co nejlepší konfiguraci pro Hadoop v HPC prostředí.

## 1.1 Struktura práce

Technologii Hadoop a její principy si popíšeme v sekci 2. Součástí této sekce je krátký popis vzniku Hadoopu a popis jeho architektury. V následující sekci 3 si popíšeme jak

[1] High Performance Computing - výpočty prováděné na vysoce výkonných počítačích, tzv. superpočítače.

Hadoop nastavit pro první spuštění a jak s ním dále pracovat. Součástí bude praktická ukázka s využitím přibalené MapReduce úlohy pro počítání slov. V sekci 4 budou popsány vybrané frameworky, které dokáží vytvářet MapReduce úlohy a využívat tak všech možností, které Hadoop poskytuje. Těmito vybranými frameworky jsou Oozie (kapitola 4.1), Hive (kapitola 4.2) a HUE (kapitola 4.3).

V další části práce se zaměříme na Hadoop v HPC prostředí. V sekci 5 bude popsán rozdíl mezi klasickou HPC architekturou a architekturou Hadoopu. Následně bude popsán framework myHadoop (kapitola 5.1), který použijeme v navazující kapitole, kde budeme spouštět Hadoop na superpočítači Anselm. Součástí této sekce je popis frameworku Mahout (kapitola 5.2), který použijeme pro testování výkonu, konkrétně budeme využívat algoritmus spektrálního shlukování. Výsledky testů jsou popsány v kapitole 5.3, kde je popsáno, jak můžeme Hadoop konfigurovat a jak Hadoop dokáže v HPC prostředí využívat přidělené zdroje. Shrnutí technologie Hadoop a provedených testů se nachází v sekci 6.

# 2 Technologie Hadoop

Celým názvem *Apache Hadoop* je open-source software, který slouží pro uložení a následné zpracování velkého množství dat. Jak již bylo řečeno v sekci 1, Hadoop se snaží poskytnout velkou výpočetní sílu za použití běžně dostupných počítačů.

## 2.1 Historie

Hadoop začal jako podprojekt projektu s názvem Nutch, který byl zase podprojekt projektu Apache Lucene[2]. Všechny tři tyto projekty založil a vedl Doug Cutting.

Nutch byl velice ambiciózní projekt, který měl využít jádra Lucene pro prohledávání webu. V rámci projektu byly implementovány algoritmy pro parsování HTML stránek, web crawlery, algoritmy pro vytváření sítě odkazů a mnoho dalších komponent důležitých pro prohledávání webu. Kromě těchto přidaných komponent se Nutch od Lucene lišil hlavně v počtu zpracovávaných dokumentů; zatímco Lucene zpracovával pouze tisíce, maximálně milióny dokumentů, Nutch při procházení webu sesbíral miliardy dokumentů, které musel následně projít a prohledat. Toho měl docílit za použití běžného hardware. Aby toho dosáhl, potřeboval vrstvu, která by dokázala práci rozdělit mezi více počítačů. Tato myšlenka sama o sobě nebyla nikterak nová, již Grace Hopper[3] v období druhé světové války řekla[1]:

*„In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."*

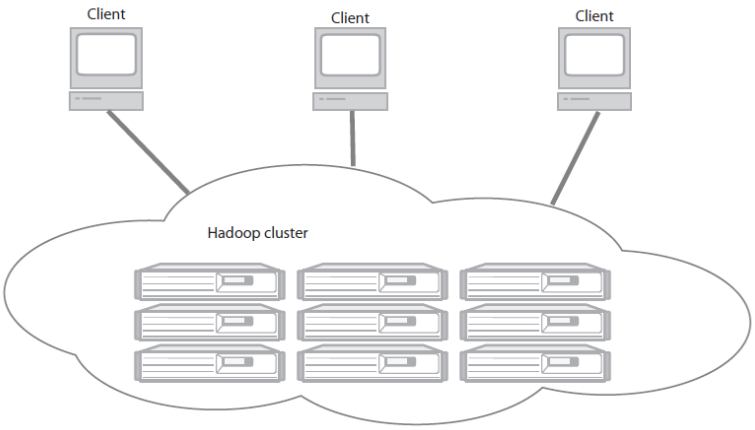
Což ve volném překladu znamená: *„Pokud jsme v minulosti používali na tažení zátěže jednoho vola, a zvětšili jsme mu zátěž, nesnažili jsme se vypěstovat většího vola, ale použili jsme více volů."*

Stejnou myšlenku použijeme ve světě IT - nebudeme používat jeden stroj, kterému zvyšujeme výkon, ale práci se snažíme distribuovat mezi více strojů. Tato myšlenka je v jádru velice jednoduchá, ale její realizace byla mnohem obtížnější.

V roce 2004 Google vydal publikaci, ve které popisoval Google File System (GFS) a MapReduce framework. Doug Cutting viděl použitelnost těchto dvou technologií pro Nutch a hned je implementoval. Nová implementace okamžitě zvýšila rozsah, ve kterém byl Nutch schopen pracovat. Mohl nyní zpracovávat milióny stránek a běžet na desítkách strojů zároveň. Doug si uvědomil, že může tuto vrstvu na zpracování dat na více strojích oddělit a tím v roce 2005 vzniknul Hadoop. V roce 2006 byl Doug najat společnosti Yahoo!, pod kterou se Hadoop nadále vyvíjel jako open-source software. O dva roky později dosáhnul Hadoop označení *Apache Top Level Project*. V roce 2008 potom Yahoo! oznámilo, že spouští Hadoop na 10 000+ Linuxových strojích pro procházení a analyzování webu. Tím se Hadoop posunul na úroveň superpočítačů.

[2]Software určený pro vyhledávání v textu - http://lucene.apache.org/.

[3]Vědecká IT pracovnice ve službách amerického námořnictva, mimo jiné autorka prvního kompilátoru pro programovací jazyk.




Obrázek 1: Hadoop cluster[3]

## 2.2 Architektura Hadoop

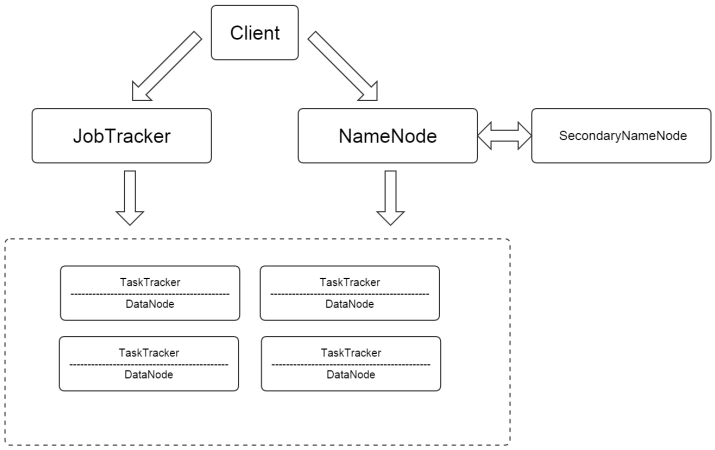
Problematika distribuovaných výpočtů je rozsáhlá a způsobů, jak je řešit, je více. Hlavní výhody Hadoopu jsou:

- Přístupnost - kdokoliv si může doma spojit pár počítačů a vytvořit si tak výpočetní cluster za pomocí běžně dostupného HW, popř. jsou volně dostupné clustery od Amazonu a dalších.
- Robustnost - protože je Hadoop navržen tak, aby běžel na běžném HW, počítá se taky s častými výpadky a Hadoop umí většinu těchto výpadků řešit bez ztráty dat.
- Škálovatelnost - Hadoop můžeme používat stejně jednoduše na jednom stroji jako na tisících, kdykoliv můžeme přidávat nebo odebírat clustery podle toho, jaký výkon potřebujeme.
- Jednoduchost - díky Hadoopu můžeme snadno psát úlohy, které se budou řešit paralelně.

Na Obrázku 1 vidíme, jak uživatel komunikuje s Hadoopem. Samotný Hadoop cluster je tvořen propojením běžného HW. Cluster obsahuje datové úložiště a můžeme proto říct, že je vytvořen „cloud“. Různí uživatelé pak mohou posílat na cluster své úlohy - tzv. *jobs*. Což je další velká výhoda Hadoopu, protože samotné úlohy mají podstatně menší velikost a přenos úlohy k datům trvá podstatně méně, než kdybychom posílali data k připraveným úlohám. V literatuře[3] se můžeme dočíst, že se jedná o tzv. „move-code-to-data philosophy“. Jedná se prakticky pouze o to, že se snažíme vyhnout pomalému přesunu dat po síti.

Hlavní moduly, na které můžeme Hadoop rozdělit jsou:

- Hadoop core - hlavní knihovny hadoopu, které obsahují veškerou logiku




Obrázek 2: Architektura Hadoop clusteru

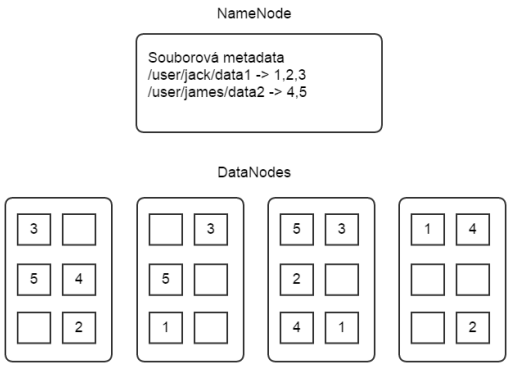
- HDFS - Hadoop Distributed File System - souborový systém, který Hadoop používá pro udržení dat rozdělených po clusterech, více popsáné v kapitole 2.2.2
- MapReduce - distribuované zpracování úloh, více v kapitole 2.2.3
- YARN - Yet Another Resource Negotiator je rozdělovač úloh, který se objevuje až u novějších verzí Hadoopu (verze Hadoop 2.0+)

### 2.2.1 Architektura Hadoop clusteru

Hadoop cluster je rozložen na další jednotlivé uzly (anglicky node). Uzly v clusteru pracují na principu master/slave, kdy dva hlavní uzly (NameNode a JobTracker) řídí práci pro ostatní podřízené uzly. Klient pošle svůj požadavek zároveň na NameNode a JobTracker. Tyto dva uzly pak rozdělí práci mezi své podřízené uzly (DataNode a TaskNode). Ukázku tohoto rozložení vidíme na Obrázku 2.

**2.2.1.1 NameNode:** Na NameNode uzlu jsou uchovávána metadata o HDFS. Slouží jako vstupní bod pro HDFS a udržuje informace jak přistoupit k dalším datům, mapuje rozložení mezi DataNode a udržuje si aktuální informace o tom, kde se jaký blok dat nachází. Protože jsou tato data kritická pro běh HDFS, vytváří NameNode svou kopii - SecondaryNameNode, ten slouží čistě jako záloha NameNode pro případný výpadek. Kdyby došlo k výpadku a ztrátě dat z NameNode, je provedena obnova uzlu pomocí dat z SecondaryNameNode, kde by se měly nacházet aktuální informace o datech.

**2.2.1.2 DataNode:** Na každém stroji v clusteru poběží proces DataNode. Tento proces obstarává základní práci, jako je například čtení/zápis bloků do HDFS. Jakmile chceme zapsat soubor do HDFS, NameNode nám tento soubor rozloží na bloky a rozešle je jednotlivým DataNode, které je teprve zapíší. DataNode mohou komunikovat mezi sebou,



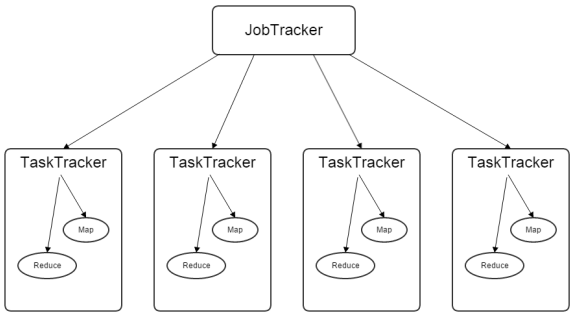

Obrázek 3: Rozložení dat mezi DataNode

aby si rozeslaly data pro požadovanou redundanci. Data jsou v HDFS záměrně redundantní, aby se předešlo ztrátě informací. Ukázku rozložení dat můžeme vidět na Obrázku 3. Vidíme, že NameNode určí, kam se jaký blok dat zapíše a rozešle data mezi DataNode. Zároveň si můžeme povšimnout velké redundance, která slouží k tomu, aby při vypadnutí jednoho DataNode mohl jeho práci zastat jiný uzel.

**2.2.1.3 JobTracker:** JobTracker se stará o rozdělení práce mezi své podřízené - TaskTracker. Jakmile klient pošle svůj kód Hadoopu ke zpracování, přijímá ho JobTracker a rozdělí práci (Job) na menší úlohy (Task). Pokud vykonání některé z úloh selže, JobTracker úlohu pošle jinému uzlu na zpracování. JobTracker se v Hadoop clusteru většinou vyskytuje jen jednou.

**2.2.1.4 TaskTracker:** Jak již bylo řečeno, i výpočetní jednotky Hadoopu dodržují master/slave architekturu. JobTracker dohlíží na vykonání úlohy jako celku, jednotlivé TaskTracker pak vykonávají samotnou práci. Každý TaskTracker je zodpovědný pouze za úlohu, kterou má aktuálně přidělenou od JobTrackeru - tato komunikace je znázorněna na Obrázku 4. Aby JobTracker věděl, že s daným TaskTrackerem probíhá komunikace správně, TaskTracker se mu hlásí pomocí tzv. signálu „heartbeat“, což je pouze signál, že komunikace mezi uzly probíhá správně. Pokud JobTracker tento signál neobdrží, bude předpokládat, že na TaskTrackeru nastala chyba a úlohu předá jinému TaskTrackeru, se kterým komunikace probíhá v pořádku.

Tím jsme si popsali klasickou topologii Hadoop clusteru, pokud se znovu podíváme na Obrázek 2, dokážeme lépe pochopit, jak Hadoop zpracovává požadavek od uživatele, a vidíme také, jak spolu jednotlivé uzly komunikují.




Obrázek 4: Komunikace mezi JobTracker a TaskTracker

### 2.2.2 HDFS

HDFS neboli Hadoop Distributed File System[4] je distribuovaný souborový systém speciálně navržený pro běh MapReduce úloh s velkým množstvím vstupních dat. Vysoká spolehlivost je zajištěna replikací dat na ostatní uzly. Běh HDFS zajišťují uzly NameNode a DataNode.

Soubor je rozložen do velkých bloků (typicky se jedná o 128 MB, ale tuto hodnotu si může uživatel nastavit pro každý soubor zvlášť). Každý tento blok je pak několikrát naklonován na DataNody (implicitně je hodnota redundance nastavena na 3, uživatel si tuto hodnotu může upravit). Každá kopie bloku je reprezentována jako dva soubory v souborovém systému, který je používán daným strojem. První soubor obsahuje samotná data a druhý soubor obsahuje metadata, včetně kontrolní sumy.

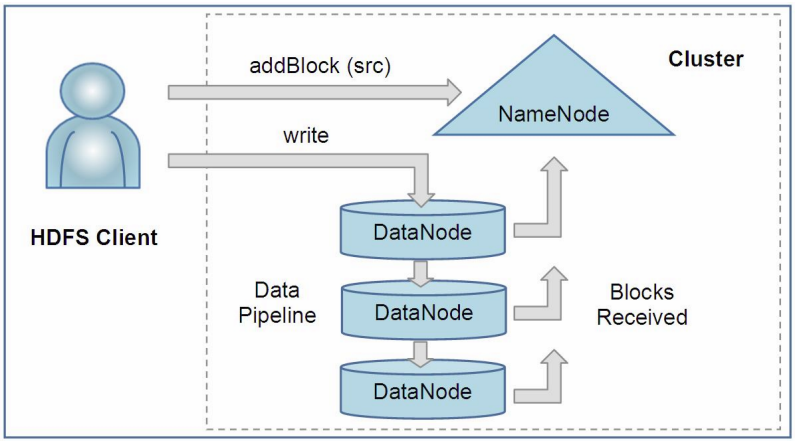
Při inicializaci HDFS každý DataNode pošle signál na NameNode, tzv. „handshake“. Tento signál ověří dostupnost uzlů a zkontroluje se *namespace ID* a *verze software*. Namespace ID je přiřazeno souborovému systému, jakmile je naformátován, toto ID se uloží na všech uzlech HDFS. Uzly s rozdílnými ID nebudou schopny mezi sebou komunikovat, což nám zaručí, že nebude narušena integrita systému.

**Poznámka 2.1** Tuto vlastnost je dobré si zapamatovat a při budoucím nastavování si uvědomit, že při každém zformátování nám bude vygenerováno nové ID. Staré soubory, které obsahují staré ID, bude potřeba smazat.

Ověření použití stejných verzí je logický krok. Pokud bychom používali rozdílné verze software, vystavujeme se riziku, že si data poškodíme, přepíšeme nebo o data přijdeme.

Jakmile se provede *handshake*, DataNode se u NameNode registrují a obdrží své unikátní ID - *storage ID*. Toto ID slouží pouze jako identifikátor daného DataNode, používá se například pro obnovení komunikace po restartu systému, změny IP adresy apod.

DataNode poté s NameNode komunikuje pomocí tzv. *block report* zpráv, kde v těchto zprávách informuje, jaké aktuálně vlastní bloky a další metadata o nich. Jakmile se dokončí tato komunikace, uzly nadále na NameNode posílají pouze *heartbeat* signály, čímž




Obrázek 5: Zápis souboru v HDFS[4]

se kontroluje komunikace mezi uzly. Implicitně se tento signál posílá každé 3 vteřiny; pokud se DataNode neozve 10 minut, NameNode považuje tento DataNode za nedostupný a všechny repliky bloků, které byly uloženy na tomto uzlu, rozešle mezi ostatní uzly, aby byla dodržena redundance.

Na Obrázku 5 vidíme postup zápisu dat. Nejprve je vytvořen nový soubor a cesta k tomuto souboru je zaslána na NameNode. Pro každý blok souboru NameNode vrátí seznam DataNodes, na kterých budou uloženy repliky tohoto bloku. Soubor je poté poslán postupně na tyto DataNode. Jakmile DataNode přijme blok, ohlásí to NameNode. Výsledné rozdělení můžeme vidět na Obrázku 3.

### 2.2.3 MapReduce

Jak již bylo zmíněno v kapitole 2.1, MapReduce je programovací model původně vyvinut společností Google. Tento model je speciálně navržen pro zpracování velkého objemu dat. MapReduce programy se dále dělí na *mappers* a *reducers*. Napsat MapReduce program je netriviální úloha, ale jakmile se nám to podaří, tak budeme schopni tento program pustit na jednom, nebo klidně i na stovkách clusterů.

MapReduce obsluhuje uzel JobTracker, který dále rozděluje práci na ostatní uzly. Vstupní data MapReduce čerpá z HDFS - tato vstupní data dále rozdělí na samostatné části a zcela paralelně části zpracuje. Výstup z mapovací funkce pak slouží jako vstup pro funkci redukční. Konečný výstup se zapíše opět na HDFS.

#### 2.2.3.1 Příklad bez použití MapReduce

Než se pustíme do MapReduce úloh, ukážeme si příklad, jak by se postupovalo bez tohoto modelu.

Jako ukázkovou úlohu použijeme klasické počítání výskytu slov ve větě. Pro naše účely nám bude stačit množina dokumentů o jednom dokumentu:

*Do as I say, not as I do.*

Pokud je množina dokumentu takto malá, můžeme rovnou napsat program, který se o počítání postará. Postačí nám program v pseudo-kódu, který vidíme ve Výpise 1.

```
define wordCount as Multiset;
    for each document in documentSet {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
display(wordCount);
```

Výpis 1: Pseudo-kód pro počítání slov[3]

V pseudo-kódu vidíme, že budeme postupně zpracovávat dokumenty z množiny všech dokumentů *documentSet*. Dále se použije funkce *tokenize()*, která nám daný dokument rozdělí na jednotlivé části. Tyto části se nazývají tokeny, v našem případě se jedná o rozdělení na jednotlivá slova - oddělujícím znakem je mezera. Nakonec si pomocí funkce *display()* zobrazíme výsledek výpisem do kozole.

Pokud bychom program spustili, dostali bychom se k výsledku, který je znázorněn v Tabulce 1.

| Slovo | Počet |
|---|---|
| as | 2 |
| do | 2 |
| I | 2 |
| not | 1 |
| say | 1 |

Tabulka 1: Četnosti slov ve větě *Do as I say, not as I do.*

Tento program splňuje zadání, avšak uspokojivě bude fungovat pouze na malé množině dokumentů. Pokud bychom tuto funkci chtěli využít v praxi, například u spam filtru, prohledání tisíců až miliónů dokumentů by bylo extrémně časově náročné. Proto bychom mohli množinu rozdělit na menší části a ty pak paralelně zpracovat na více počítačích. V první fázi by každý počítač prošel pouze část původní množiny, v druhé části by se výsledky od všech předchozích počítačů spojily a vznikl by konečný výstup. Pseudokód pro první fázi je znázorněn ve Výpise 2.

```
define wordCount as Multiset;
for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
sendToSecondPhase(wordCount);
```

Výpis 2: První fáze počítání slov[3]

První část je téměř identická s původním programem (viz. Výpis 1). Pouze místo celé množiny dokumentů *documentSet* projdeme pouze její podmnožinu - *documentSubset*. Ve druhé fázi spojíme výsledky dohromady, tato část je znázorněna ve Výpise 3.

```
define totalWordCount as Multiset;
for each wordCount received from firstPhase {
    multisetAdd (totalWordCount, wordCount);
}
```

Výpis 3: Druhá fáze počítání slov[3]

Tento program by však fungoval pouze v ideálních podmínkách, stále zůstává spousta problémů, které mohou nastat. Jak efektivně rozdělit data? Pokud si je budou jednotlivé počítače stahovat z centrálního úložiště, pak narazíme na to, že přenos po síti u centrálního prvku se bude chovat jako úzké hrdlo. Řešením by bylo rozdělit dokumenty na jednotlivé stroje. Další problém, na který můžeme narazit, je nedostatek operační paměti. Pokud počítáme pouze spisovná slova, nejspíš bude velikost RAM běžného počítače dostatečná. Pokud však začneme započítávat veškerá slangová slova a překlepy, dostaneme se za hranice možností naší RAM. Nemusíme počítat pouze slova, mohou to být výskyty IP adres v logu apod. Dřív nebo později bychom dospěli k závěru, že potřebujeme výsledky zapisovat na disk; v takovém případě bychom museli napsat mnohem komplexnější program.

Dále v druhé části stále přetrvává problém, že bude práci muset obstarat pouze jeden stroj. Pokud budeme mít hodně počítačů v první části, v druhé části nás bude zpomalovat nedostačující výkon jednoho stroje. Nejspíš nás hned napadne rozdělení práce ve druhé fázi opět na více strojů. Tím se dostáváme k dalším problémům - jak spočítat konečný výskyt jednoho slova, když budeme počítat paralelně na několika strojích? Konečný výsledek musíme opět rozdělit na části, například bychom mohli rozdělit práci tak, aby každý stroj počítal pouze slova, která začínají určitým písmenem. V první fázi bychom si data rovnou připravovali vytvářením kolekcí *wordCount-a*, *wordCount-b* atd. Tím by nám vzniklo 26 kolekcí (pokud počítáme pouze anglickou abecedu). V druhé fázi by se jednotlivé kolekce rozeslaly na Počítač A, Počítač B atd.

Tím jsme si jednoduchou úlohu, jako je počítání slov, značně zesložitili. Pokud bychom chtěli podobně distribuovanou práci využít v praxi, musíme se připravit na chyby, výpadky disků apod. Proto je mnohem výhodnější použít framework jako Hadoop. Pokud napíšeme program v MapReduce modelu, Hadoop se postará o veškeré distribuování práce.

**2.2.3.2 Příklad s použitím MapReduce** MapReduce programy běží ve dvou hlavních fázích, tzv. *mapping* a *reducing*. Každá fáze má svou funkci, která ji obstarává - tyto funkce se nazývají *mapper* a *reducer*. Nejprve si vstupní data převezme *mapper*, zpracuje je a svůj výstup pošle na vstup funkce *reducer*, který je zpracuje a pošle na konečný výstup. Jednoduše řečeno, v první fázi si data rozdělíme a uspořádáme, v druhé fázi je zpracujeme. Můžeme si povšimnout prakticky totožného postupu jako v kapitole 2.2.3.1.

Podobnost není náhodná, protože MapReduce framework vychází ze zkušeností s prací na distribuovaných programech, kde se převážně používá rozdělení na části.

Protože budeme využívat modelu MapReduce, který je generický a nezáleží na typu vstupních dat, potřebujeme vhodnou datovou strukturu, na kterou vstupní data převedeme. MapReduce používá list párů klíč/hodnota (*key/value pairs*). Klíč a hodnota jsou základních datových typů jako integer nebo string.

Jak již bylo zmíněno, při psaní MapReduce programu píšeme dvě funkce - *map* a *reduce*. V Tabulce 2 můžeme vidět vstupy (input) a výstupy (output) jednotlivých funkcí.

| | Input | Output |
|---|---|---|
| map | <k1, v1> | list(<k2, v2>) |
| reduce | <k2, list(v2)> | list(<k3, v3>) |

Tabulka 2: Vstupy a výstupy pro MapReduce

Vstup pro aplikaci je list (<klíč, hodnota>). Na první pohled by se mohlo zdát, že bude složité vstupní data do tohoto formátu převést. V praxi je to však velice primitivní a používají se páry jako `list(<String fileName, String file_content>)`, kde se jako klíč použije název souboru a jako hodnota se použije obsah souboru. Nebo pokud máme jeden velký soubor, jako například log soubor, použijeme `list(<Integer line_number, String log_event>)`, tedy jako klíč nám poslouží číslo řádku v log souboru a jako hodnota pak bude záznam z tohoto řádku.

Tento list je rozdělen na jednotlivé páry, které slouží jako argumenty pro mapovací funkci. V Tabulce 2 jsme mohli vidět, že vstupní dvojice `<k1, v1>` je přetransformována mapovací funkcí na `list(<k2, v2>)`. V praxi se často nepracuje s klíčem `k1` a zpracovává se pouze hodnota `v1`, zpracování a výsledná transformace záleží na implementaci mapovací funkce. Pokud se vrátíme k našemu příkladu, ve kterém nás zajímá četnost slov, můžeme si dovolit ignorovat název souboru a zaměřit se pouze na jeho obsah. Uvnitř mapovací funkce vytvoříme `list(<String word, Integer 1>)`. Nyní máme dvě možnosti jak k počítání slov přistoupit. Můžeme vytvářet páry a následně vždy zvětšovat počet výskytů, takže kdybychom například měli slovo *hadoop* v textu použito 5× vznikne nám pár `<"hadoop", 5>`, nebo můžeme mít pár `<"hadoop", 1>` v listu vložený 5×. Druhý postup je jednodušší na implementaci. Použití prvního postupu by mohlo program zrychlit, ale do ladění a zrychlování se můžeme pustit až po lepším pochopení MapReduce.

Výstup všech mapovacích funkcí se nakonec spojí do jednoho velkého listu obsahujícího `<k2, v2>`. Všechny páry, které mají stejný klíč, se spojí do nového páru `<k2, list(v2)>`. V našem případě si můžeme představit, že máme dvě stanice v clusteru. Každá stanice má funkce *map* a *reduce*. Funkci *map* na první stanici můžeme označit jako *Mapper 1* a na druhé stanici jako *Mapper 2*. Potom pokud jsme z Mapperu 1 dostali slovo *hadoop* 5 × a z Mapperu 2 jsme dostali slovo *hadoop* 2×, výsledný pár, který bude sloužit jako vstupní argument pro redukční funkci, bude `<"hadoop", list(1,1,1,1,1,1,1>`. Což je konečný počet, kolikrát se slovo vyskytlo v celé množině dokumentů. Naše redukční funkce pak už pouze vytvoří čitelnější list hodnot `<k3, v3>`, kde `k3` bude slovo a

`v3` jeho konečná četnost. Celý pseudo-kód této MapReduce funkce je zobrazen ve Výpise 4. Jak již bylo řečeno, vstupem pro aplikaci je list (<klíč, hodnota>), tento list si předem rozdělíme na jednotlivé páry a vstup pro funkci *map* pak bude už jen jeden konkrétní pár.

```
map(String filename, String document) {
    List<String> T = tokenize(document);
    for each token in T {
        emit ((String)token, (Integer) 1);
    }
}

reduce(String token, List<Integer> values) {
    Integer sum = 0;
    for each value in values {
        sum = sum + value;
    }
    emit ((String)token, (Integer) sum);
}
```

Výpis 4: Pseudo-kód pro MapReduce funkci[3]

**Poznámka 2.2** Funkce `emit()` použita v pseudo-kódu je funkce obsažená v MapReduce frameworku. Tato funkce slouží pro postupné generování prvků v listu, usnadňuje tak programátorům práci s rozsáhlými listy.

# 3 Hadoop v praxi

V této sekci bude popsáno jak připravit prostředí pro Hadoop a následně jak jej nastavit pro první spuštění. Dále bude vysvětleno jak na Hadoopu spustit předpřipravené úlohy i jak tyto úlohy upravit a vytvořit tak vlastní.

## 3.1 Příprava prostředí

### Operační systém - Linux

Pro spuštění Hadoopu pouze s jedním clusterem nám postačí mít nainstalovaný OS Linux pouze na jednom stroji. V našem případě byl použit Linux Mint 16, který běžel jako virtuální stroj.

### Instalace Javy

Pro spuštění Hadoopu je potřebné mít nainstalovanou Javu ve verzi 1.6 a vyšší. V našem případě budeme používat verzi 1.7.0_67.

Stáhneme balíček požadované verze Javy. V našem případě budeme využívat 64bit verzi Javy.

1. Nejprve si vytvoříme složku pro Javu:

```
sudo mkdir -p -v /opt/java/64
```

2. Jdeme do složky se staženým balíčkem Javy, rozbalíme a přesuneme do připravené složky:

```
cd ~/Downloads
tar -zxvf jre-7u67x64.tar.gz
sudo mv -v jre1.7.0_67 /opt/java/64
```

3. Posledním krokem je nastavit nové JRE jako výchozí:

```
sudo update-alternatives --install "/usr/bin/java"  "java"
"/opt/java/64/jre1.7.0_67/bin/java" 1

sudo update-alternatives --set java /opt/java/64/jre1.7.0_67/bin/java'' 1
```

## 3.2 Instalace a konfigurace Hadoop

Základní požadavky pro běh systému Hadoop, jako je operační systém a instalace Javy, jsme si popsali v předcházejících krocích. Nyní nakonfigurujeme systém tak, abychom mohli nainstalovat Hadoop. K tomu budeme potřebovat uživatele, který bude mít všechna práva a bude moci instalovat, konfigurovat a spouštět všechny služby. Hadoop vyžaduje SSH přístup ke správě jednotlivých stanic v clusteru. Pro tyto účely se generuje RSA klíč s prázdným heslem, aby při každé interakci Hadoopu se stanicí nebylo vyžadováno heslo.

Hadoop podporuje tři základní režimy[9]:

- samostatný režim
- pseudo-distribuovaný režim
- distribuovaný režim

V samostaném režimu beží Hadoop jako jediný Java proces. Tento režim nepodporuje práci se souborovým systémem HDFS. Je vhodný zejména pro vytváření a ladění jednoduchých MapReduce úloh.

V pseudo-distribuovaném režimu Hadoop pracuje pouze s jedním strojem; tento stroj zajišťuje funkci výpočetního uzlu i datového úložiště zároveň. Z toho vyplývá, že na tomto jednom stroji jsou spuštěny všechny potřebné služby - NameNode, SecondaryNameNode, JobTracker, DataNode a TaskTracker. Tento režim nám plně simuluje rozložení zátěže a můžeme s Hadoopem provádět všechny operace. V tomto režimu si ukážeme následující ukázku spuštění Hadoopu.

Posledním režimem je distribuovaný režim. Tento režim odpovídá úplné konfiguraci Hadoop clusteru. Obsahuje master stanici, která slouží pro hostování služeb NameNode , SecondaryNameNode a JobTracker a minimálně jednu další stanici, na které běží slave služby - DataNode a TaskTracker. V tomto režimu už je zátěž plně distribuována.

### 3.2.1 Hadoop v pseudo-distribuovaném režimu

Před instalací Hadoopu se ujistíme, že máme správně nainstalovanou Javu, o tom se můžeme přesvedčit příkazem:

```
java -version
```

Pokud máme správně nainstalovanou Javu, tak se nám zobrazí výpis její verze, pokud ne, vrátíme se do kapitoly 3.1.

#### 3.2.1.1 Vytvoření uživatele Hadoop

Pro běh se doporučuje použít zvlášť vytvořeného uživatele z důvodu oddělení instalace Hadoop od jiných softwarových aplikací. Nejprve vytvoříme novou skupinu pro uživatele, skupina bude mít název *hadoop*:

```
addgroup hadoop
```

Následně vytvoříme uživatele *hadoopuser*:

```
adduser -ingroup hadoop hadoopuser
```

Zadáme heslo pro účet a můžeme vyplnit podrobnější nastavení.

#### 3.2.1.2 Konfigurace SSH

Jak již bylo zmíněno, tak hadoop pro komunikaci vyžaduje SSH. Pro sestavení clusteru v pseudo-distribuovaném režimu je v tomto případě nutné nakonfigurovat SSH přístup na localhost (na kterém nám poběží Hadoop) pro uživatele *hadoopuser*.

Pro přepnutí na uživatele *hadoopuser* použijeme následující příkaz:

```
su hadoopuser
```

Pomocí následujícího příkazu vytvoříme RSA klíč s prázdným heslem:

```
ssh-keygen -t rsa -P
```

Potvrdíme uložení klíče do složky */home/hadoopuser/.ssh/id_rsa*. Zobrazí se nám vygenerovaný otisk klíče v alfanumerické a obrázkové podobě.

Pro přidání do seznamu autorizovaných klíču na localhost použijeme následující příkaz:

```
cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

Naše nastavení si můžeme otestovat příkazem:

```
ssh localhost
```

Pokud dostáváme chybové hlášení, pravděpodobně nám chybí openssh-server, nainstalujeme jej a příkaz spustíme znovu:

```
sudo apt-get install openssh-server
```

#### 3.2.1.3 Instalace Hadoop

V době psaní této práce je aktuální verze Hadoop 2.5.0 (vydáno 14. srpna 2014). Pro naše účely použijeme stabilní verzi Hadoop 1.2.1, pro kterou je přizpůsoben taky myHadoop, který budeme používat v další fázi práce (kapitola 5.1).

Stáhneme si balíček s požadovanou verzí Hadoopu[4] a rozbalíme jej příkazem:

```
tar -xvf hadoop-1.2.1.tar.gz
```

Název se samozřejmě bude lišit podle verze, kterou jsme zvolili. Balíček je vhodné rozbalit do domovského adresáře námi vytvořeného *hadoopuser*.

**Poznámka 3.1** Pokud se rozhodnete použít verzi Hadoop 2.0 a vyšší, je nutné si uvědomit, že se změnila struktura a soubory budou rozmístěny ve složkách jiným způsobem, než je popsáno dále v práci.

#### 3.2.1.4 Konfigurace Hadoop

Nejprve musíme Hadoop navést k instalaci Javy. Toho docílíme tak, že definujeme lokální proměnnou JAVA_HOME. Přejdeme do složky s nastavením `~/hadoop-1.2.1/conf` a otevřeme soubor `hadoop-env.sh`. Najdeme řádek, který začíná *export JAVA_HOME =* a doplníme cestu. V našem případě bude řádek vypadat následovně (všimneme si odstraněného znaku # - což značí začátek komentáře):

```
export JAVA_HOME=/usr/lib/jvm/java-7openjdk-amd64
```

Před konfigurací Hadoopu si vytvoříme složku s názvem *single* pro ukládání vstupních a výstupních dat. Provedeme tak pomocí příkazu:

```
mkdir single
```

Složku vytváříme do složky `home/hadoopuser`, následně přidáme práva uživateli *hadoopuser*:

[4] http://hadoop.apache.org/releases.html

```
chown hadoopuser:hadoop /home/hadoopuser/single
```

Hadoop nastavujeme pomocí tří souborů, a to: *core-site.xml*, *mapred-site.xml*, *hdfs-site.xml*.

Začneme úpravou *core-site.xml*, otevřeme pro úpravu například příkazem:

```
vim conf/core-site.xml
```

V tomto souboru se definuje složka pro soubory vytvářené Hadoopem, název souborového systému a cesta, která obsahuje host a port, kde běží služba NameNode. Upravíme soubor na tuto podobu:

```
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/home/hadoopuser/single/hadoop-hadoopuser</value>
  </property>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

Výpis 5: Výpis souboru core-site.xml

Dále upravíme soubor *mapred-site.xml*. V tomto souboru se definuje host a port, na kterém běží instance JobTracker. V našem případě bude spouštěn pouze jeden proces s jedinou MapReduce úlohou. Otevřeme soubor:

```
vim conf/mapred-site.xml
```

A editujeme na následující podobu:

```
<configuration>
  <property>
    <name>mapred.job.tracker</name>
    <value>localhost:9001</value>
  </property>
</configuration>
```

Výpis 6: Výpis souboru mapred-site.xml

Dále definujeme počet replikací v souboru *hdfs-site.xml*. Implicitně nastavená hodnota faktoru replikace je 1 - tedy žádná replikace. Protože používáme pouze jednu stanici, necháme tuto hodnotu. Otevřeme soubor:

```
vim conf/hdfs-site.xml
```

A editujeme na následující podobu:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

Výpis 7: Výpis souboru mapred-site.xml

#### 3.2.1.5 Spuštění Hadoop

Před prvním spuštěním je nutné souborový systém HDFS naformátovat. Přejdeme do složky, kde máme Hadoop a spustíme příkaz:

```
bin/hadoop namenode -format
```

Poté se spustí všechny služby příkazem:

```
bin/start-all.sh
```

Výpis všech běžících služeb získáme příkazem:

```
jps
```

Výpis bude vypadat následovně:

```
15277 DataNode
16356 Jps
15476 JobTracker
15116 NameNode
15608 TaskTracker
15396 SecondaryNameNode
```

Podrobný výpis o dostupných stanicích (v našem případě pouze jedna) se dá získat pomocí příkazu:

```
bin/hadoop dfsadmin -report
```

Výpis bude vypadat podobně:

```
Configured Capacity: 13654904832 (12.72 GB)
Present Capacity: 8019025920 (7.47 GB)
DFS Remaining: 8018984960 (7.47 GB)
DFS Used: 40960 (40 KB)
DFS Used%: 0%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Datanodes available: 1 (1 total, 0 dead)

Name: 127.0.0.1:50010
Decommission Status : Normal
Configured Capacity: 13654904832 (12.72 GB)
DFS Used: 40960 (40 KB)
Non DFS Used: 5635878912 (5.25 GB)
DFS Remaining: 8018984960(7.47 GB)
DFS Used%: 0%
DFS Remaining%: 58.73%
Last contact: Sat Aug 16 13:06:50 CEST 2014
```

## 3.3 Spuštění úlohy na Hadoopu

V následujících kapitolách si ukážeme spuštění MapReduce úlohy na Hadoopu a taky postup, jak bychom mohli tuto úlohu dále ladit a vylepšovat.

### 3.3.1 Ukázková úloha WordCount

Pro otestování v pseudo-distribuovaném režimu použijeme zkušební program *WordCount*, který je přidán v každém balíku Hadoop. Tento program pomocí MapReduce procesu spočítá četnost slov ve vstupním textovém souboru.

Hadoop přijímá data jen ze svého souborového systému (HDFS), nejprve je tedy potřeba vytvořit složku pro vstupní data:

```
bin/hadoop fs -mkdir input
```

Se souborovým systémem pracujeme pomocí sady příkazů, které jsou velice podobné těm linuxovým, další příkazy jsou například:

```
bin/hadoop fs -ls
bin/hadoop fs -cat
bin/hadoop fs -mkdir
bin/hadoop fs -put
bin/hadoop fs -copyFromLocal
bin/hadoop fs -copyToLocal
bin/hadoop fs -rm
bin/hadoop fs -rmr
bin/hadoop fs -tail
bin/hadoop fs -chmod
```

Dále vytvoříme textový soubor, například pomocí příkazu:

```
vim inputtext.txt
```

Do textového souboru vložíme text, ve kterém budeme chtít zjistit počet slov. V našem případě je to následující text:

*Lorem ipsum dolor sit amet, reque appareat evertitur ex mei, ex facer pertinax conceptam vim. Ubique graeci alienum at eos, ei alii essent interesset has. Commodo laboramus has id, ea cum voluptaria vituperata. Odio meis blandit cu eam, illud graece vocibus sea at. No nullam reprimique per, cum cu eligendi dissentiunt.*[5]

Vytvořený soubor přesuneme do vstupní složky pomocí příkazu:

```
bin/hadoop fs -put inputtext.txt input/
```

Tím máme vše připraveno pro spuštění úlohy. Implementaci úlohy *Wordcount* nalezneme v příloze A, spustíme ji příkazem:

```
bin/hadoop jar hadoop-examples-1.2.1.jar wordcount input output
```

Zobrazí se nám podrobný výpis o zpracování úlohy. V něm uvidíme následující postup:

```
14/08/16 13:21:53 INFO mapred.JobClient: Running job: job_201408161205_0001
14/08/16 13:21:54 INFO mapred.JobClient:  map 0% reduce 0%
14/08/16 13:22:09 INFO mapred.JobClient:  map 100% reduce 0%
14/08/16 13:22:26 INFO mapred.JobClient:  map 100% reduce 100%
14/08/16 13:22:31 INFO mapred.JobClient: Job complete: job_201408161205_0001
```

Při spuštění úlohy jsme tedy zadali soubor, kde je naprogramována úloha, název této úlohy, vstupní a výstupní složky. Ve vstupní složce nesmí být nic jiného než soubory, které chceme zpracovávat. Výstupní složka nesmí být vytvořena, Hadoop si ji vytvoří v průběhu zpracování. O tom se můžeme přesvědčit příkazem:

[5] Text ze stránky `http://www.lipsum.com/`

```
bin/hadoop fs -ls
```

Zobrazí se nám tento výpis:

```
Found 2 items
drwxr-xr-x   - root supergroup          0 2014-08-16 13:20 /user/root/input
drwxr-xr-x   - root supergroup          0 2014-08-16 13:22 /user/root/output
```

Abychom se podívali, co se nám vytvořilo ve složce *output*, spustíme příkaz:

```
bin/hadoop fs -ls output/
```

Následný výpis:

```
-rw-r--r--   1 root supergroup          0 2014-08-16 15:57 /user/root/output/_SUCCESS
drwxr-xr-x   - root supergroup          0 2014-08-16 15:56 /user/root/output/_logs
-rw-r--r--   1 root supergroup        194 2014-08-16 15:57 /user/root/output/part-r-00000
```

Vidíme, že se nám vytvořily tři soubory, které obsahují záznam o vykonané úloze. Nás zajímá soubor s názvem *part-r-00000*, který obsahuje výsledek, na který se můžeme podívat pomocí příkazu:

```
bin/hadoop fs -cat output/part-r-00000
```

Tímto by se nám vypsal výsledek, zde pouze ve zkrácené verzi:

```
Commodo 1
Lorem 1
...
vocibus 1
voluptaria 1
```

### 3.3.2 Vytvoření vlastní úlohy pro Hadoop

V předchozí kapitole 3.3.1 jsme spustili předpřipravenou verzi úlohy WordCount. Ve výsledném výpise jsme si mohli všimnout, že je tato úloha značně nedokonalá a jako oddělovač používá pouze mezeru. V následující kapitole tuto úlohu upravíme, zkompilujeme a spustíme.

Nejprve si připravíme pracovní adresáře, se kterými budeme dále pracovat:

```
mkdir workspace
mkdir workspace/src
mkdir workspace/classes
```

Do vytvořené složky *src* nakopírujeme soubor se zdrojovám kódem programu *WordCount.java*:

```
cp src/examples/org/apache/hadoop/examples/WordCount.java workspace/src/WordCount.java
```

Před samotnou úpravou tuto kopii zkompilujeme a spustíme:

```
javac -classpath hadoop-core-1.2.1.jar:lib/commons-cli-1.2.jar -d ~/workspace/classes/
~/workspace/src/WordCount.java

jar -cvf ~/workspace/wordcount.jar -C ~/workspace/classes/ .
```

Pro ověření funkčnosti se nově zkompilovaný program spustí:

```
bin/hadoop jar ~/workspace/wordcount.jar org.apache.hadoop.examples.WordCount input/
output/
```

**Poznámka 3.2** Složka *output* nesmí existovat, takže ji nezapomeneme odstranit, případně pojmenujeme složku jiným názvem.

Pro editaci programu otevřeme *WordCount.java* a můžeme upravovat zdrojový kód. Následně vše zkompilujeme a spustíme v prostředí Hadoop.

# 4 Hadoop ekosystém

Hadoop sám o sobě slouží pouze jako prostředek pro vykonání „těžké práce“ neboli výpočtů. Abychom mohli používat Hadoop, musíme napsat MapReduce úlohy[6]. Psát tyto úlohy je zdlouhavé a náročné. Proto vznikla spousta frameworků zaměřující se na zjednodušení psaní těchto úloh. Vznikaly také nové způsoby spravování HDFS, protože se při velkém množství dat stává nepřehledným. Nakonec začaly vznikat komplexní nástroje, které zastřešovaly a spojovaly použití ostatních frameworků. Na ilustračním obrázku 6 můžeme vidět vypsány některé z frameworků, které dokáží využít Hadoop. Nástrojů pro Hadoop je opravdu hodně, v této sekci budou popsány pouze některé z nich.

## 4.1 Oozie

Apache Oozie je open-source webový nástroj pro Hadoop, který slouží pro plánování a koordinaci jednotlivých úloh. Původně byl vyvinut pro správu složitých úloh ve společnosti Yahoo![12]. Příklad jednoduché úlohy můžeme vidět na obrázku 7. Nejprve se zpracuje MapReduce úloha, následně Pig[7] a nakonec se výsledek vypíše pomocí Hive[8] dotazu. Hive a jeho dotazování je vysvětleno v následující kapitole 4.2. Oozie spolupracuje přímo se službou JobTracker, díky tomu úloha bude rovnoměrně distribuována mezi všechny uzly.

Plány úloh na sebe mohou navazovat, mohou být spouštěny manuálně nebo mít nastaven časovač spuštění. Plány se zapisují do XML souboru pomocí jazyka zvaného hPDL - Hadoop Process Definition Language[9]. Jako příklad takového zápisu můžeme uvést:

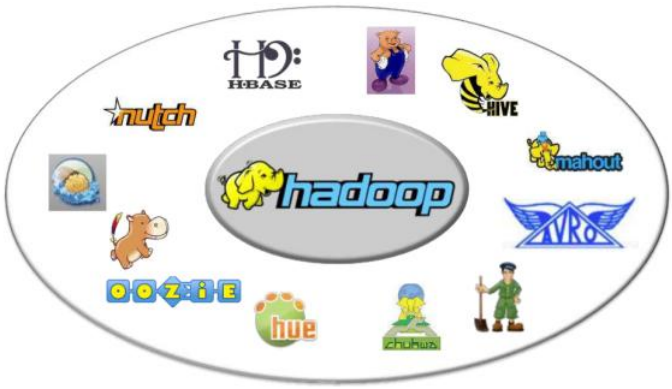
```
<workflow-app name='test-wf' xmlns="uri:oozie:workflow:0.1">
    <start to ='test'/>

    <action name='test'>
        <map-reduce>
        ...
        <map-reduce>
        <ok to='end'>
    </action>

    < kill name='kill'>
        <message>
        ...
        <\message>
    </ kill >

    <end name='end'/>
    </workflow-app>
```

[6] Nově ted' také YARN úlohy, označovány jako MR2, jakožto přímý nástupce MapReduce - MR1

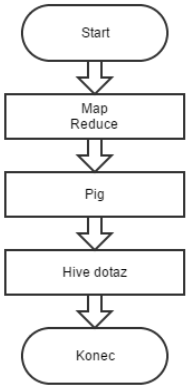
[7] Další z frameworků z Hadoop ekosystému, sloužící pro analýzu a zpracování dat, více na `https://pig.apache.org/`.

[8] Technolgie Hive slouží pro zpracování dat na HDFS, více v kapitole 4.2.

[9] Jazyk navržen speciálně pro Oozie, podobný jazyku XML.




Obrázek 6: Hadoop ekosystém[11]




Obrázek 7: Příklad Oozie plánované úlohy

Výpis 8: Ukázka Oozie úlohy[12]

Ukázka plánování je pouze příkladem zápisu a je nefunkční, slouží jen pro demonstraci způsobu zapisování. Jsou zde vidět kontrolní uzly jako *start* a *kill*, které slouží pro řízení toku zpracování, a pak *action* uzly, které slouží pro zpracovávané úlohy. Podrobný popis jazyka hPDL nalezneme na stránkach s dokumentací [10].

### 4.1.1 Spuštění Oozie

Když máme obecné chápání o Oozie, můžeme se jej pokusit spustit. Problematikou sestavení a spuštění Oozie se zabývá následující kapitola, která slouží jako návod.

[10]https://oozie.apache.org/docs/3.2.0-incubating/WorkflowFunctionalSpec.html#OozieWFSchema

V našem případě použijeme verzi Oozie 4.4.0[11]. Stáhneme a rozbalíme balíček do libovolné složky. Pokud používáme jinou verzi Javy než implicitně nastavenou verzi 1.6, před sestavením upravíme verzi v souboru *pom.xml*, který se nachází v rozbalené složce Oozie.

```
<javaVersion>1.7</javaVersion>
<targetJavaVersion>1.7</targetJavaVersion>
```

Z výše uvedených řádků je zřejmé, že v našem případě použijeme verzi Javy 1.7. Po upravení verze Javy můžeme Oozie sestavit pomocí připraveného skripku *mkdistro.sh*, zůstaneme v hlavní Oozie složce a zadáme příkaz:

```
bin/mkdistro.sh
```

**Poznámka 4.1** Při používání Oozie verze 4.0.0 je velice častá chyba při sestavení:

```
[ERROR] Failed to execute goal on project oozie-hadoop: Could not resolve dependencies
for project org.apache.oozie:oozie-hadoop:jar:2.2.0-SNAPSHOT.oozie-4.0.0: Could not find
artifact org.apache.hadoop:hadoop-client:jar:2.2.0-SNAPSHOT in apache.snapshots.repo
```

Tato chyba je vyřešena v novejší verzi Oozie, která v době testování nebyla k dispozici. Pokud chybu chceme vyřešit a zůstat u verze 4.0.0, vyhledáme si všechny výskyty výrazu *2.2.0-SNAPSHOT* v souborech *pom.xml*:

```
 grep -l "2.2.0-SNAPSHOT" `find . -name "pom.xml"`
```

Výsledkem budou následující soubory:

```
./pom.xml
./hadooplibs/hadoop-distcp-2/pom.xml
./hadooplibs/hadoop-2/pom.xml
./hadooplibs/hadoop-test-2/pom.xml
```

V těchto souborech vyhledáme výraz *2.2.0-SNAPSHOT* a upravíme na *2.2.0*.

Po úspěšném sestavení se nám zobrazí výpis podobný tomuto:

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2:58.061s
[INFO] Finished at: Wed Oct 15 18:01:19 CEST 2014
[INFO] Final Memory: 232M/626M
[INFO] ------------------------------------------------------------------------

Oozie distro created, DATE[2014.10.15-15:58:19GMT] VC-REV[unavailable], available at
[/home/hadoopuser/oozie-4.0.0/distro/target]
```

Z výpisu je pro nás nejdůležitější informace, že se sestavení podařilo a jeho umístění. Přejdeme do této složky a rozbalíme následující soubor:

```
tar -zxvf oozie-4.0.0-distro.tar.gz
```

V nově vytvořené složce *oozie-4.0.0* vytvoříme složku *libext*

```
mkdir libext
```

[11] http://archive.apache.org/dist/oozie/4.0.0/

Pokud používáme verzi Hadoopu, která je obsažena ve složce *oozie-4.0.0/hadooplibs*, překopírujeme soubory ze složky s touto verzí do právě vytvořené složky *oozie-4.0.0/libext*. V našem případě používáme verzi Hadoop 1.2.1, která v *hadooplibs* obsažena není. Do složky *libext* překopírujeme soubory typu *jar* z naší domovské Hadoop složky:

```
sudo cp *.jar /home/hadoopuser/oozie-4.0.0/distro/target/oozie-4.0.0/libext/
sudo cp lib/*.jar /home/hadoopuser/oozie-4.0.0/distro/target/oozie-4.0.0/libext/
```

Další knihovny, které potřebujeme do naší *libext* složky, jsou z balíčku *ext-2.2.zip*[12].

Nyní si připravíme Hadoop pro použití s Oozie. Toho docílíme úpravou souboru *hadoop-1.2.1/conf/core-site.xml*, do tohoto souboru přidáme následující vlastnosti:

```
<property>
   <name>hadoop.proxyuser.[OOZIE_SERVER_USER].hosts</name>
   <value>[OOZIE_SERVER_HOSTNAME]</value>
</property>
<property>
   <name>hadoop.proxyuser.[OOZIE_SERVER_USER].groups</name>
   <value>[USER_GROUPS_THAT_ALLOW_IMPERSONATION]</value>
</property>

V~nasem pripade bude konfigirace vypadat takto:

<property>
   <name>hadoop.proxyuser.hadoopuser.hosts</name>
        <value>localhost</value>
</property>
<property>
        <name>hadoop.proxyuser.hadoopuser.groups</name>
        <value>hadoop</value>
</property>
```

Výpis 9: Nastavení Oozie v souboru core-site.xml

Nyní máme vše potřebné pro sestavení *war* souboru. Přejdeme do složky *oozie-4.0.0/distro/target/oozie-4.0.0-distro/oozie-4.0.0* a v této složce spustíme:

```
bin/oozie-setup.sh prepare-war
```

Pokud vše proběhne v pořádku, objeví se podobný výpis:

```
New Oozie WAR file with added 'ExtJS library, JARs' at
/home/hadoopuser/oozie-4.0.0/distro/target/oozie-4.0.0/oozie-server/webapps/oozie.war
INFO: Oozie is ready to be started
```

Dále si připravíme databázi pomocí příkazu:

```
bin/oozie-setup.sh db create -run
```

Pokud vše proběhlo v pořádku, zobrazí se výpis:

```
   setting CATALINA_OPTS="$CATALINA_OPTS -Xmx1024m"

Validate DB Connection
DONE
Check DB schema does not exist
```

[12] extjs.com/deploy/ext-2.2.zip

```
DONE
Check OOZIE_SYS table does not exist
DONE
Create SQL schema
DONE
Create OOZIE_SYS table
DONE

Oozie DB has been created for Oozie version '4.0.0'

The SQL commands have been written to: /tmp/ooziedb-6291881397510192198.sql
```

Nyní máme vše připraveno pro spuštění Oozie 4.0.0:

```
bin/oozied.sh start
```

Abychom nemuseli stále opisovat URL adresu, na které běží Oozie, otevřeme si soubor *oozie* a soubor editujeme:

```
vim bin/oozie
...
export OOZIE_URL="http://localhost:11000/oozie/";
```

Nyní můžeme odzkoušet, zda nám Oozie v pořádku běží:

```
bin/oozie admin -status
...
System mode: NORMAL
```

Protože s Oozie nebudeme dále pracovat a jen budeme chtít otestovat funkčnost, použijeme pro tento účel předpřipravené ukázkové úlohy. Ty nalezneme v balíčku *Oozie-4.0.0/oozie-examples.tar.gz*, rozbalíme jej:

```
tar -zxvf oozie-examples.tar.gz
```

Dále otevřeme soubor *examples/apps/map-reduce/job.properties* a upravíme porty, v našem případě to bude vypadat následovně:

```
nameNode=hdfs://localhost:9000
jobTracker=localhost:50030
queueName=default
examplesRoot=examples
```

Uložíme a složku s příklady nahrajeme na HDFS, abychom k nim mohli přistupovat přes Hadoop. Pro nahrání souboru na HDFS musíme mít spuštěnou službu NameNode.

```
bin/hadoop fs -put ~/oozie-4.0.0/distro/target/oozie-4.0.0/examples examples
```

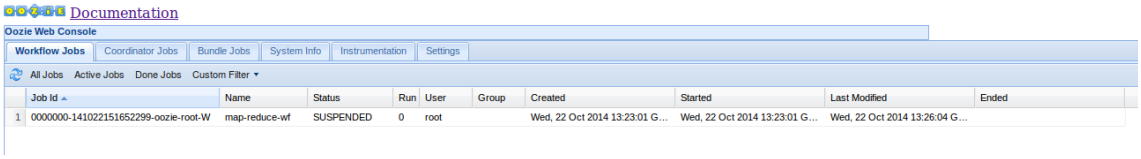
Ukázkovou úlohu spustíme příkazem:

```
bin/oozie job -oozie http://localhost:11000/oozie -config
     examples/apps/map-reduce/job.properties -run
```

Tím dostaneme naše *job ID*:

```
job: 0000000-141022151652299-oozie-root-W
```

Pomocí tohoto ID můžeme s úlohou pracovat. Například si můžeme zobrazit informace o úloze, kterou jsme spustili:




Obrázek 8: Webové rozhraní pro Oozie 4.0.0

```
bin/oozie job -oozie http://localhost:11000/oozie -info 0000000-141022151652299-oozie-root-W
...
flow Name : map-reduce-wf
App Path      : hdfs://localhost:9000/user/root/examples/apps/map-reduce
Status        : SUSPENDED
Run           : 0
User          : root
Group         : -
Created       : 2014-10-22 13:23 GMT
Started       : 2014-10-22 13:23 GMT
Last Modified : 2014-10-22 13:26 GMT
Ended         : -
CoordAction ID: -

Actions
------------------------------------------------------------------------------------------------
ID                                                 Status    Ext ID      Ext Status Err Code
------------------------------------------------------------------------------------------------
0000000-141022151652299-oozie-root-W@:start:       OK        -           OK         -
------------------------------------------------------------------------------------------------
0000000-141022151652299-oozie-root-W@mr-node   START_MANUAL -           -
JA009 ------------------------------------------------------------------------------------------
```

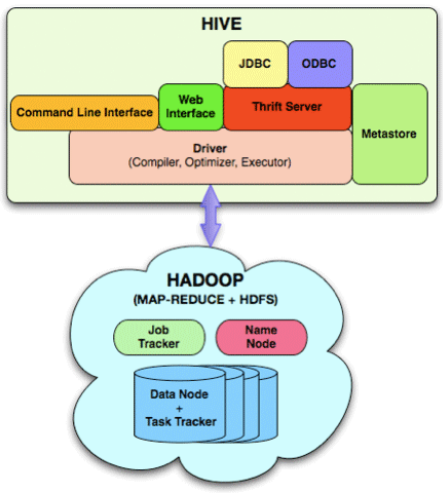
Zde vidíme vypsány všechny potřebné informace. Pokud se chceme podívat na Oozie v grafické podobě, můžeme k webovému rozhraní přistoupit přes port 11000, na kterém je Oozie implicitně nastaveno. A protože nám Oozie běží na localhostu, výsledná adresa v našem případě bude *http://localhost:11000/oozie*. Jak takové webové rozhraní vypadá, můžeme vidět na obrázku 8.

Webové rozhraní nabízí další nástroje na řízení úloh apod. Jako rozšíření pro Oozie můžeme použít komplexní nástroj HUE, který je popsán v kapitole 4.3.3.

## 4.2 Hive

Apache Hive je open-source datový sklad. Vznikl pro manipulaci s daty pro Hadoop, je tedy přizpůsoben práci s velkými daty. Jako první začal s vývojem Facebook[14]. V dnešní době jej používají pro správu svých rozsáhlých dat i další velké společnosti jako Netflix, Amazon a další.

Hive má tři hlavní funkce - sumarizaci dat, dotazování nad daty a analýzu dat. Za tímto účelem využívá svůj dotazovací jazyk *HiveQL* (Hive Query Language), který automaticky překládá dotazy, které vychází ze syntaxe jazyka SQL, do MapReduce úloh, které jsou následně zpracovány Hadoopem[14]. Hive působí jako klasické relační SŘBD, jsou zde však zásadní rozdíly. Protože je Hive spouštěno nad Hadoopem, který vykonává




Obrázek 9: Apache Hive architektura[13]

časově náročné úlohy, můžeme očekávat, že Hive nebude dodávat výsledky okamžitě, ale s velkým zpožděním. Z toho vyplývá, že se Hive nehodí pro použití, kdy potřebujeme okamžitou reakci, jak by bylo očekáváno u databází jako DB2[13] a jiné. Hive je tzv. „read-based“, není tedy vhodný pro transakční zpracování.

Na obrázku 9 můžeme vidět architekturu. Je patrné, že se Hive skládá z dalších dílčích částí:

- **JDBC, ODBC** – Java Database Connectivity a Open Database Connectivity, jedná se o API pro přístup k ostatním databázím
- **Command Line Interface** – jak název napovídá, jedná se o modul, který zprostředkovává práci s Hive přes příkazovou řádku
- **Web Interface** – modul, který zprostředkovává grafické webové rozhraní
- **Thrift server** – základní server, který slouží pro přístup uživatelů, založen na Apache Thrift[14]
- **Metastore** – modul sloužící pro správu metadat
- **Driver** – modul, který překládá HiveQL dotazy na úlohy pro Hadoop

[13] http://www-01.ibm.com/software/data/db2/
[14] http://thrift.apache.org/

### 4.2.1 HiveQL

HiveQL je založen na jazyku SQL, ale nedodržuje plně standard SQL-92[15]. Nabízí rozšíření, které SQL neobsahuje, například vkládání záznamů do více tabulek zároveň. Naopak má omezenou práci s indexy[13], obsahuje ale i specifičtější typy indexů jako bitmapový index. Příkazy pro vkládání, editování a mazání splňují ACID[16] od verze Hive 0.14. Překladač následně tento příkaz přeloží do MapReduce úloh, v nových verzích je samozřejmostí YARN podpora.

### 4.2.2 Spuštění Hive

V této kapitole se opět budeme zabývat praktickou ukázkou. V našem případě použijeme Apache Hive verze 0.13.1. Pro otestování spuštění si můžeme stáhnout již zkompilovanou verzi. Stažený balíček si rozbalíme a změníme práva příslušnému uživateli. V průběhu ukázky budeme používat HDFS, je tedy zřejmé, že je potřeba mít po celou dobu spuštěný Hadoop.

Jako ukázkovou úlohu budeme řešit problém, kdy chceme zjistit počet vydání dané knihy. Jako dataset použijeme „Book crossing dataset“[17].

**Poznámka 4.2** Data nalezneme také na přiloženém DVD.

Pokud si data zobrazíme, zjistíme, že na prvním řádku se nachází hlavička, kde jsou popsány jednotlivé sloupce, na dalších řádcích jsou samotná data. Nejprve si data upravíme pro naše použití:

```
sed 's/&amp;/&/g' BX-Books.csv | sed -e '1d' | sed 's/;/$$$/g' |
sed 's/"$$$"/";"/g' > BX-BooksCorrected.csv
```

Předešlý příkaz vyhledá všechny znaky „ ; “ (středník) v samotném obsahu a nahradí je znaky „$$$“. Tuto změnu děláme proto, protože později budeme středník používat jako oddělovač sloupců. Dále změní znak „&“ na pouhé „&“. Nakonec odstraní hlavičku souboru, abychom ji nezpracovávali jako součást dat, což hlavička není. Hlavičku je ovšem dobré zkopírovat a zálohovat si pro budoucí použití. Poté upravený soubor nahrajeme na HDFS:

```
bin/hadoop fs -mkdir input
bin/hadoop fs -put /home/hadoopuser/bookssets/BX-BooksCorrected.csv input
```

Před samotným spuštěním Hive musíme připravit složky, kde si bude Hive uchovávat svá data, složkám také nastavíme potřebná práva:

[15] Stantard z roku 1992, který dodržuje jazyk SQL - `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt`.

[16] Atomicity (Atomicita), Consistency (Konzistence), Isolation (Izolovanost), Durability (Trvalost) - souhrn vlastností, zajišťující spolehlivost databáze.

[17] https://github.com/Orzota/tutorials/blob/master/Resources/BX-CSV-Dump.zip

```
hadoop fs -mkdir /tmp
hadoop fs -mkdir /user/hive/warehouse
hadoop fs -chmod g+w /tmp
hadoop fs -chmod g+w /user/hive/warehouse
```

Jako poslední krok nastavení je potřeba v souboru Hive nastavit cestu k Hadoopu. Otevřeme soubor s názvem *hive* a přidáme cestu:

```
vim bin/hive
export HADOOP_HOME=/home/hadoopuser/hadoop-1.2.1
```

Nyní máme vše připraveno pro spuštění a odzkoušení Hive. Spustíme jej příkazem:

```
bin/hive
...
hive>
```

Jak vidíme, zobrazí se nám příkazová řádka Hive pod označením *hive*>. Na tuto příkazovou řádku můžeme psát příkazy v jazyce HiveQL4.2.1. Abychom mohli pracovat s daty, potřebujeme stejně jako v relačních databázích tabulku. Tu si vytvoříme následovně:

```
hive> CREATE TABLE IF NOT EXISTS BXDataSet
    > (ISBN STRING,
    > BookTitle STRING,
    > BookAuthor STRING,
    > YearOfPublication STRING,
    > Publisher STRING,
    > ImageURLS STRING,
    > ImageURLM STRING,
    > ImageURLL STRING)
    > COMMENT 'BX-Books Table'
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '\;'
    > STORED AS TEXTFILE;
```

Můžeme si povšimnout, že zde použijeme dříve zálohovanou hlavičku souboru. Taky stojí za zmínku velká podobnost HiveQL s jazykem SQL. Následně načteme do tabulky data z HDFS:

```
hive> LOAD DATA INPATH '/user/root/input/BX-BooksCorrected.csv' OVERWRITE INTO TABLE
BXDataSet;
```

Tímto jsme si připravili vše potřebné pro dotazování se nad daty. Pokud se vrátíme k původnímu problému a budeme chtít zjistit počet vydání pro každou knihu, stačí použít příkaz:

```
hive> select yearofpublication, count(booktitle) from bxdataset group by
yearofpublication;
```

Opět si můžeme povšimnout velké podobnosti s jazykem SQL, ze kterého HiveQL vychází. V pozadí se však spouští MapReduce úloha, která příkaz zpracovává. Výsledek tohoto dotazu bude podobný:

```
"2038" 1
"2050" 2
Time taken: 28.663 seconds, Fetched: 116 row(s)
```

Vidíme, že pro tak malý výsledek, jako je 116 řádků, je téměř půl minuty opravdu dlouhá doba. Musíme si však uvědomit, že Hive není určeno pro takto malá data a předpokládá se určité zpoždění při vykonání dotazu, jak již bylo popsáno v úvodu této kapitoly.

Jako poslední část si připravíme Hive pro použití s dalším frameworkem, kterým je HUE (kapitola 4.3). Abychom v HUE mohli Hive využívat, spustíme *Hiveserver*:

```
bin/hiveserver2
```

Dále je potřeba nastavit příslušná oprávnění na dočasné soubory, které si Hive vytváří tak, aby k nim mělo HUE pozdější přístup.

```
bin/hadoop fs -chmod -R 777 /tmp
```

## 4.3 HUE

HUE (Hadoop User Experience)[16] je sdružení několika webových aplikací, které usnadňují práci s Hadoop clusterem. HUE umožňuje procházet HDFS, spravovat úlohy, procházet Hive tabulky a využívat mnoho dalších frameworků z celého Hadoop ekosystému. HUE běží jako webová aplikace a není teda potřeba žádná instalace na straně klienta. Velice tímto zjednodušuje přístup k samotnému Hadoop clusteru, kdy se uživatel pouze přihlásí přes svůj prohlížeč a získá tak okamžitý přístup k Hadoopu a všem službám, které má v HUE nastavené. Z tohoto jasně vyplývá zaměření projektu HUE - zjednodušit přístup a složitost Hadoopu, odstranění příkazové řádky a podobně.

### 4.3.1 Architektura HUE

Architektura HUE je velice jednoduchá, jak je vidět na obrázku 10; jedná se o pouhého zprostředkovatele mezi ostatními službami. Uživatel zadá svůj požadavek na HUE a tento požadavek je následně předán dál příslušnému frameworku k zpracování.

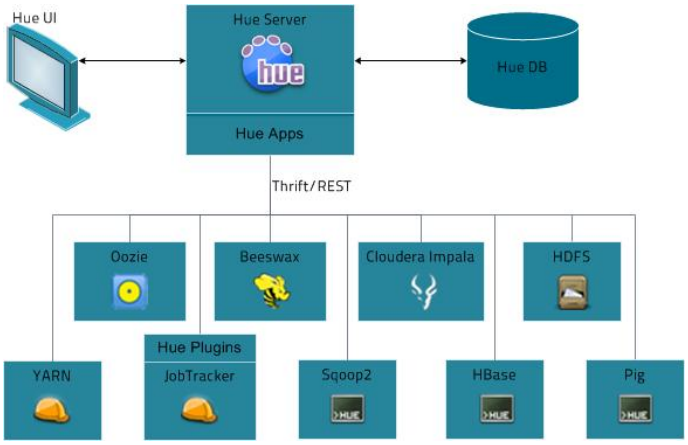
Jak již z názvu vyplývá, jedná se o „user experience“, tedy něco, co má zpříjemnit uživateli samotné používání. Nejlépe si tedy HUE vysvětlíme na praktické ukázce, na které také uvidíme rozdíly oproti dřívějšímu používání Hadoopu.

### 4.3.2 Spuštění HUE

Opět začneme stažením příslušných souborů potřebných pro následná nastavení. Můžeme tak učinit pomocí příkazu:

```
sudo apt-get install hue hue-server
```

Následně bude potřeba aktualizace pluginu HUE:




Obrázek 10: Architektura HUE[15]

```
sudo apt-get install hue-plugins
```

V další části potřebujeme připravit Hadoop pro použití s HUE. Začneme úpravou souboru *hdfs-site.xml*, kde povolíme WebHDFS pro NameNode a DataNode přidáním následující vlastnosti:

```
<property>
  <name>dfs.webhdfs.enabled</name>
  <value>true</value>
</property>
```

Výpis 10: Nastavení HUE v souboru hdfs-site.xml

Jako další upravíme soubor *core-site.xml*, kde povolíme všechny uživatele a skupiny, aby mohli HUE používat. V další části bychom zde mohli nastavit určitá omezení, nyní nám však stačí povolit HUE pro všechny uživatele:

```
<property>
  <name>hadoop.proxyuser.hue.hosts</name>
  <value>*</value>
</property>
<property>
  <name>hadoop.proxyuser.hue.groups</name>
  <value>*</value>
</property>
```

Výpis 11: Nastavení HUE v souboru core-site.xml

Nakonec provedeme samotné nastavení HUE. HUE se nastavuje pomocí souboru */etc/hue/conf/hue.ini*. V tomto souboru nalezneme veškerá nastavení, které budeme potřebovat. Prozatím se zaměříme na sekci Hadoop, zde doplníme naše umístění:

```
hadoop_home=/home/hadoopuser/hadoop-1.2.1
```

Dále specifikujeme adresu a port, na které nám běží HDFS, neboli NameNode:

```
fs_defaultfs=hdfs://localhost:9000
```

Veškerá nastavení uložíme a nyní můžeme HUE spustit:

```
sudo /etc/init.d/hue start
...
* Starting Hue for Hadoop  hue
[ OK ]
```

Webserver HUE v základním nastavení běží na portu 8888, pokud chceme tento port změnit, můžeme tak učinit v souboru *hue.ini*. Připojit se na HUE můžeme na adrese *localhost:8888*. Při prvním připojení budeme vyzváni,abychom si vytvořili účet, tento účet bude následně používán jako administrátorský účet. V samotném HUE moc možností využití není - můžeme zde procházet HDFS a nahrávat soubory přes webové rozhraní. Stejně tak si můžeme zobrazit aktivní Hadoop úlohy. Abychom mohli plně používat HUE, potřebujeme nastavit další nástroje. Nastavíme si proto již dříve vysvětlené Oozie a Hive.

### 4.3.3 Oozie na HUE

Oozie bylo již dříve vysvětleno v kapitole 4.1. Následující kapitola obsahuje pouze doplňující informace o tom, jak nastavit Oozie v prostředí HUE.

Pro použití HUE prostředí pro Oozie nejdříve nahrajeme soubory ze složky *sharelib* na HDFS, toho docílíme následovně:

```
bin/hadoop fs -put /home/hadoopuser/oozie-4.0.0/sharelib/ /user/oozie/share
```

Poté složku přejmenujeme. Protože HUE v základním nastavení hledá cestu *share/lib* namísto použitého *sharelib*:

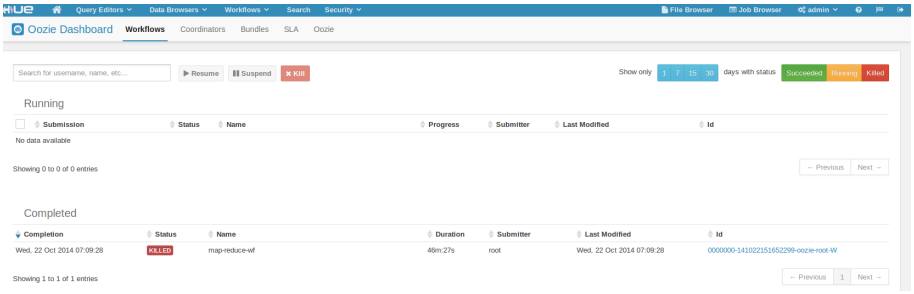
```
sudo bin/hadoop fs -mv /user/oozie/share/sharelib /user/oozie/share/lib
```

Dále nastavíme Oozie, aby mohl správně komunikovat s HUE. Nastavení provedeme v souboru */etc/oozie/conf/oozie-site.xml*. Přidáme následující vlastnosti:

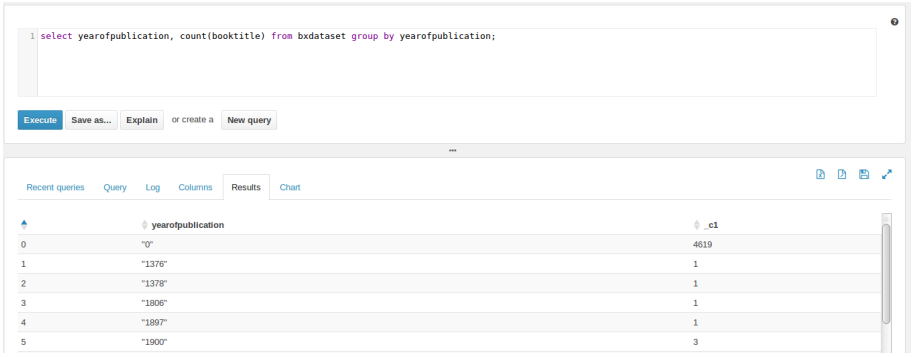
```
<property>
    <name>oozie.service.ProxyUserService.proxyuser.hue.hosts</name>
    <value>*</value>
</property>
<property>
    <name>oozie.service.ProxyUserService.proxyuser.hue.groups</name>
    <value>*</value>
</property>
```

Výpis 12: Nastavení Oozie pro HUE

Nakonec zkontrolujeme, jestli nám nastavení v souboru */etc/hue/conf/hue.ini* v části Oozie souhlasí s naším nastavením. Výsledkem těchto nastavení bude stav, kdy přes webové rozhraní můžeme procházet a upravovat běžící úlohy na Hadoopu pomocí HUE rozhraní. Jak to bude ve výsledku vypadat, se můžeme podívat na obrázku 11.




Obrázek 11: Oozie v prostředí HUE




Obrázek 12: Hive v prostředí HUE

### 4.3.4 Hive na HUE

Pokud máme nastaveno Hive tak, jak bylo popsáno v kapitole 4.2, máme vše připraveno pro použití na HUE. Stačí zkontrolovat soubor */etc/hue/conf/hue.ini*, zda souhlasí nastavení a máme volné porty, které chceme používat. V základním nastavení se Hive odkazuje na localhost, toto nastavení ponecháme. Poté se přes HUE můžeme připojit k Hive a vyzkoušet stejný příklad jako v kapitole 4.2.2. Přepneme se do Hive rozhraní a do vyhrazeného textového pole zadáme náš příkaz, jak můžeme vidět na obrázku 12. Výsledek se nám vypíše do přehledné tabulky.

## 4.4 Shrnutí

V předešlých kapitolách jsme si ukázali jen základní nástroje pro práci s HDFS a se správou Hadoop úloh. Existují i další nástroje, které zvládnou generovat MapReduce úlohy pro Hadoop. Tyto nástroje zastřešuje HUE, který je nejspíš nejkomplexnější nástroj ze všech. Můžeme v něm psát i jednoduché skripty v Pythonu, například pro rychlou úpravu dat.

Dalším typem nástrojů pro Hadoop jsou knihovny s různými algoritmy pro analýzu dat. Hlavním zástupcem v této oblasti je Mahout, který si popíšeme a odzkoušíme v ka-

pitole 5.2. Apache Mahout obsahuje například shlukovací algoritmy, které následně budeme spouštět také v HPC prostředí, abychom Hadoop otestovali z výkonostní stránky.

# 5 Hadoop v HPC

Většina klasických HPC zdrojů se skládá z výpočetních uzlů s minimálním lokálním úložištěm. Tyto uzly jsou pak mezi sebou propojeny ve vysokorychlostní síti a následně napojeny na výkonný paralelní souborový systém jako *Lustre* nebo IBM *General Parallel File System (GPFS)*[6]. V tom je podstatný rozdíl oproti Hadoopu, kde každý výpočetní uzel má svoje lokální úložiště, na které si ukládá svá data a výsledky - tato architektura se nazývá *shared-nothing-architecture*[6]. Porovnání těchto dvou architektur vidíme na Obrázku 13.

Shared-nothing architektura je navržena pro použití na běžně dostupném HW a propojení klasických PC, která zastávají jak výpočetní, tak i datovou úlohu. Všechny uzly jsou mezi sebou propojeny a data jsou sdílena pomocí HDFS (2.2.2).

Z toho vyplývají dvě hlavní překážky pro implementaci Hadoopu do HPC prostředí:

- zajistit spolupráci mezi rozdělováním Hadoop úloh a rozdělovačem úloh v HPC prostředí
- implementovat shared-nothing architekturu na tradiční HPC architekturu

Jak již bylo popsáno v kapitolách 2.2, Hadoop používá shared-nothing architekturu, ale vždy jeden uzel určí jako hlavní uzel (Master) a ostatní uzly jako podřízené (Slaves). Na hlavním uzlu spustí NameNode, které řídí HDFS a JobTracker, které řídí rozdělování a zpracování úloh. Ostatní uzly poté hostují DataNode a TaskTracker, tyto typy uzlů jsou podrobněji popsány v kapitole 2.2. Budeme tedy potřebovat označit jeden z přiřazených uzlů jako Master a ostatní uzly jako Slave.

## 5.1 myHadoop

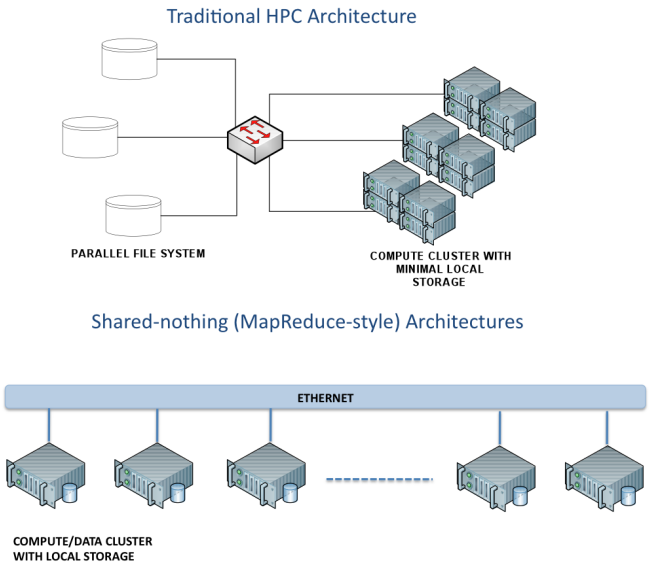
Framework myHadoop byl vytvořen v superpočítačovém centru v San Diegu[6]. Jeho hlavním účelem je spustit Hadoop v HPC prostředí s použitím standardních plánovačů úloh, jako je TORQUE nebo SGE. Jedná se o open-source projekt, jehož zdrojové kódy jsou volně ke stažení [18].

### 5.1.1 myHadoop architektura

Na Obrázku 14 můžeme vidět celkovou arcitekturu myHadoop na HPC zdrojích. Uživatelé mohou využít myHadoop k alokování Hadoop clustrů tzv. na-vyžádání. Nejprve vyžádáme potřebné zdroje od HPC zdrojového manažera a nastavíme, jaké uzly budou master a jaké slave. Nakonec nakonfigurujeme a spustíme Hadoop na příslušných alokovaných uzlech. Poté mohou uživatelé spouštět své Hadoop úlohy a po vykonání těchto úloh je Hadoop zastaven a všechny zdroje jsou dealokovány, tedy uvolněny.

Na Obrázku 14 jsme si mohli povšimnout, že myHadoop může pracovat ve dvou různých módech - *nepersistentní* a *persistentní* mód (non-persistent a persistent na Obrázku 14).

[18] https://github.com/glennklockwood/myhadoop/




Obrázek 13: HPC a Shared-nothing architektura[6]

V nepersistentním módu je Hadoop nakonfigurován tak, aby využíval lokální úložiště. Tento mód má však dva nedostatky. Nejprve je tu problém s velikostí lokálního úložiště, které nemusí postačovat. Poté je zde problém s alokací zdrojů - nikdy nemáme jistotu, že nám budou alokovány ty uzly, na kterých jsou na lokálním úložišti uloženy výsledky úlohy, tudíž by pro nás byly tyto výsledky nedostupné. Abychom tyto problémy vyřešili, můžeme použít persistentní mód, kde jdou data uložena v distribuovaném souborovém systému jako je Lustre nebo GPFS.

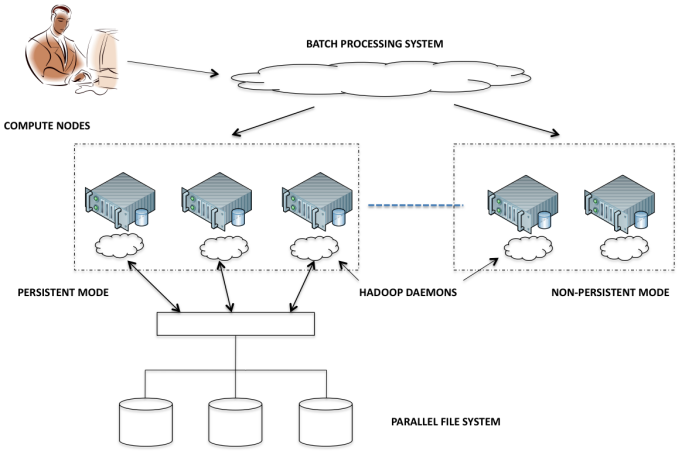
### 5.1.2 myHadoop implementace

Framework myHadoop byl původně navržen pro Hadoop verze 0.20.2. V této práci použijeme patch pro práci s Hadoopem 1.2.1. Hlavní myšlenkou myHadoopu je, aby si každý uživatel mohl vygenerovat takovou konfiguraci, kterou potřebuje[6]. Tato konfigurace by pak měla sloužit ostatním uživatelům ke spouštění Hadoop úloh bez potřeby konfigurace a administrátorských (root) práv[19].

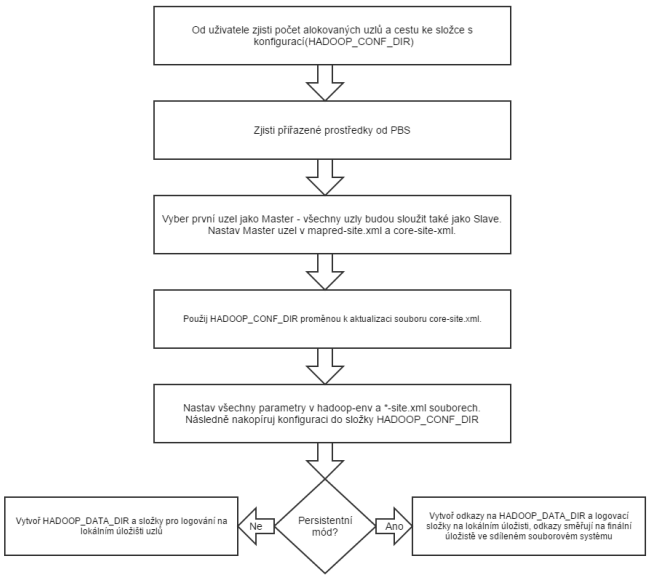
Jak myHadoop postupuje, můžeme vidět na Obrázku 15. Jednotlivé soubory, které myHadoop nastaví, jsou:

- **masters**: V tomto souboru je zapsán uzel, ktery slouží jako Master. Na tomto uzlu je spuštěno NameNode, Secondary Namenode, JobTracker.
- **slaves**: V tomto souboru je uložen seznam všech uzlů, které slouží jako Slaves. Na těchto uzlech je spuštěno DataNode, TaskTracker.

[19] Z těchto vlastností vznikl název *myHadoop*.




Obrázek 14: myHadoop architektura[6]




Obrázek 15: Postup myHadoop

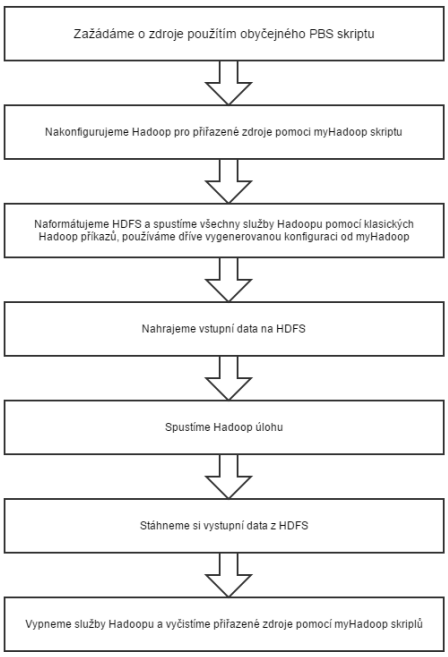
- **core-site.xml**: Hlavní konfigurační soubor, který obsahuje umístění HDFS (HADOOP_DATA_DIR) na každém uzlu a URI pro HDFS server (skládá se z adresy uzlu a portu). Dále obsahuje různé ladící parametry, pomocí kterých můžeme ovlivňovat výpočty Hadoopu, například velikost čtecí/zapisovací mezipaměti a další.
- **hdfs-site.xml**: HDFS je konfigurační soubor, který obsahuje parametry pro konfiguraci distribuovaného souborového systému, například počet replikací a velikost HDFS bloku.
- **mapred-site.xml**: Soubor sloužící pro konfiguraci MapReduce úloh. Obsahuje URI pro JobTracker, počet mapovacích a redukčních úloh, které mohou být spuštěny najednou, JAVA_OPTS pro konfiguraci JVM na uzlech.
- **hadoop-env.sh**: Script konfigurující samotné Hadoop prostředí. Důležité parametry jsou například velikost Hadoop haldy, umístění Hadoop logů, parametry pro JVM.

Jak již bylo zmíněno, na Obrázku 15 je znázorněn postup myHadoopu. Nejprve uživatel napíše skript pro PBS (myHadoop je univerzální a skripty mohou být napsány také například pro scheduler SGE, záleží na prostředí, ve kterém myHadoop spustíme). V tomto skriptu uživatel spustí skripty pro myHadoop, které vygenerují konfiguraci pro Hadoop. Jakmile se spustí myHadoop skript (konkrétně *myhadoop-config.sh*), proběhne nastavení všech potřebných souborů a konfigurace se uloží do nově vzniklého adresáře (tento adresář je zastoupen proměnnou *HADOOP_CONF_DIR*). Uživatel zvolí, jestli chce použít persistentní nebo nepersistentní mód [20]. Jakmile je vše připraveno a myHadoop konfigurační skript je spuštěn, zjišťují se nejprve zdroje, které nám byly přiděleny od PBS. Z proměnné PBS_NODEFILE[21] vezme myHadoop první uzel a nastaví jej jako *Master*. Všechny uzly včetně Master uzlu se pak zařadí do seznamu *Slave* uzlů. Tyto informace jsou zapsány do příslušných souborů (HADOOP_CONF_DIR/master a HADOOP_CONF_DIR/slaves). Skript také upraví soubory mapred-site.xml a core-site.xml - v těchto souborech se nachází jména uzlů pro HDFS (NameNode) a pro MapReduce (JobTracker). Poté se zjistí umístění, kde se nachází data z proměné HADOOP_DATA_DIR. Nastaví se všechny ladící parametry a kompletní soubory se nakonec zapíší do složky HADOOP_CONF_DIR.

Pokud uživatel spustí Hadoop v nepersistentním módu, myHadoop vytvoří složku HADOOP_DATA_DIR na všech uzlech a výsledné HDFS může být naformátováno. Pokud uživatel spustí Hadoop v persistentním módu, myHadoop vytvoří na každém uzlu pouze odkaz na výslednou složku HADOOP_DATA_DIR; výsledná složka se nachází ve sdíleném souborovém systému.

Na Obrázku 16 vidíme, jak myHadoop pracuje z pohledu uživatele. Jak již bylo řečeno, uživatel začne PBS skriptem, kterým zažádá o přidělení zdrojů. Poté uživatel nastaví Hadoop pomocí myHadoop konfiguračního skriptu. Dále už jen zbývá spustit Hadoop běžným způsobem. S Hadoopem už se nadále pracuje klasicky - nahrajeme vstupní

[20] Tyto informace se zadávají jako argumenty při spuštění skriptu
[21] Proměná prostředí PBS, která obsahuje seznam alokovaných uzlů




Obrázek 16: Postup myHadoop z pohledu uživatele

data na HDFS, spustíme Hadoop úlohy, stáhneme si data z HDFS. Jakmile jsme s prací na Hadoopu hotovi, nesmíme si zapomenout data z HDFS stáhnout, protože pokud používáme nepersistetní mód, data jsou uloženy lokálně na uzlech a při dalším spuštění nemáme jistotu, že nám budou přiřazeny stejné uzly a k datům bychom ztratili přístup, což by vedlo ke samotné ztrátě dat. Proto musíme všechna data stáhnout ještě před dealokací zdrojů. Nakonec uživatel zastaví všechny Hadoop služby a může spustit připravené čistící myHadoop skripty. Spuštění těchto skriptů není povinné, ale doporučuje se, aby na uzlech nezůstávala zbytečná data, která by pouze zabírala místo pro ostatní uživatele.

### 5.1.3 Spuštění myHadoop v HPC prostředí Anselm

Pro testování Hadoopu v HPC jsme použili infrastrukturu superpočítače Anselm[22], který spadá pod národní superpočítačové centrum IT4Inovations. Anselm má dohromady 207 výpočetních uzlů a jako správce zdrojů používá PBS a Lustre distribuovaný souborový systém.

V této práci se nebudeme zabývat samotným přístupem do HPC prostředí. Budeme předpokládat, že jej uživatel má.

[22]Kompletní hardwarou specifikaci nalezneme na `https://docs.it4i.cz/anselm-cluster-documentation/hardware-overview`


Obrázek 17: Anselm prostředí

**Poznámka 5.1** Následující postup je ověřen pouze v HPC prostředí Anselm a nemusí se shodovat s jiným HPC prostředím.

Začneme přihlášením na Anselm (Obrázek 17). Jak již víme, Hadoop potřebuje pro svůj běh Javu, načteme si tedy modul Javy pomocí příkazu:

```
module load java
```

Ověříme si verzi Javy (Hadoop požaduje verzi 1.6 a vyšší), tím si také ověříme, že se nám Java načetla:

```
java -version
```

V našem případě máme k dispozici Javu verze 1.7. Dalším krokem je opatření si balíčků s Hadoopem verze 1.2.1 a myHadoop. Balíčky nalezneme například na GitBub.com[23]. Balíčky rozbalíme do libovolných složek, v našem případě použijeme složky */Workspace/hadoop-1.2.1* a */Workspace/myHadoop*. Přejdeme do složky *hadoop-1.2.1/conf* a použijeme následující příkaz, který nám připraví Hadoop tak, aby odpovídal šabloně, kterou bude dále myHadoop využívat.

```
$ patch < ~/Workspace/myHadoop/myhadoop-master/myhadoop-1.2.1.patch
...
patching file core-site.xml
patching file hdfs-site.xml
patching file mapred-site.xml
```

**Poznámka 5.2** Složka *myhadoop-master* vznikla po rozbalení balíčku myHadoop.

Z výpisu je vidět, že myHadoop pouze upravil tři hlavní soubory, pomocí kterých se Hadoop nastavuje. Soubory není nutné dále nastavovat, ale můžeme se podívat, jak myHadoop konfiguraci provedl.

Otevřeme si soubor *core-site.xml* a obsah bude následující:

```
<property>
  <name>hadoop.tmp.dir</name>
    <value>HADOOP_TMP_DIR</value>
  <description>A base for other temporary directories.</description>
```

[23] https://github.com/glennklockwood/myhadoop

```
</property>

<property>
    <name>fs.default.name</name>
    <value>hdfs://MASTER_NODE:54310</value>
    <description>The name of the default file system.  A~URI whose
    scheme and authority determine the FileSystem implementation. The
    uri's␣scheme␣determines␣the␣config␣property␣(fs.SCHEME.impl)␣naming
␣␣␣the␣FileSystem␣implementation␣class.␣␣The␣uri's authority is used to
    determine the host, port, etc. for a filesystem.</description>
</property>
```

Výpis 13: Nastavení myHadoop v souboru core-site.xml

Vidíme, že myHadoop připravil šablonu pomocí proměnných, do kterých se bude dosazovat až po zjištění přiřazených zdrojů. Parametr *hdfs://MASTER_NODE:54310* je pro nás hlavní informace v tomto souboru, jedná se o adresu a port, na které poběží NameNode čili hlavní služba pro HDFS. V dalším souboru *hdfs-site.xml* nalezneme následující nastavení:

```
<property>
  <name>dfs.name.dir</name>
  <value>DFS_NAME_DIR</value>
  <description>Determines where on the local filesystem the DFS name node
    should store the name table.  If this is a comma-delimited list
    of directories then the name table is replicated in all of the
    directories, for redundancy. </description>
  <final>true</final>
</property>

<property>
  <name>dfs.data.dir</name>
  <value>DFS_DATA_DIR</value>
  <description>Determines where on the local filesystem an DFS data node
      should store its blocks.  If this is a comma-delimited
      list of directories, then data will be stored in all named
      directories, typically on different devices.
      Directories that do not exist are ignored.
  </description>
  <final>true</final>
</property>

<property>
  <name>dfs.replication</name>
  <value>DFS_REPLICATION</value>
  <description>HDFS is partly designed to allow storage failures and uses
      replication for this. Since either your data on myhadoop jobs is only
      supposed to live through a single run or you can use persistent data
      that will most likely run on solid hardware it is quite save to keep
      replication at 1 and reduce the IO overhead.
  </description>
</property>

<property>
```

```
  <name>dfs.block.size</name>
  <value>DFS_BLOCK_SIZE</value>
  <description>The HDFS block size defines the size of the parts in which
  the HDFS files will be divided and distributed over the data nodes.
  </description>
</property>

<property>
  <name>fs.default.name</name>
  <value>hdfs://MASTER_NODE:54310</value>
  <description>The name of the default file system. A~URI whose
  scheme and authority determine the FileSystem implementation. The
  uri's␣scheme␣determines␣the␣config␣property␣(fs.SCHEME.impl)␣naming
␣␣the␣FileSystem␣implementation␣class.␣␣The␣uri's authority is used to
  determine the host, port, etc. for a filesystem.
   </description>
</property>
```

Výpis 14: Nastavení myHadoop v souboru hdfs-site.xml

Zde se nachází cesta ke složkám, kam se budou ukládat data HDFS. Dále se pak nastavuje počet replikací dat na uzlech. V poslední části vidíme opět URL adresu, na které poběží NameNode. Tato informace slouží pro jednotlivé Slave uzly, na kterých běží DataNode. Jedná se o adresu, na kterou budou posílat tzv. *heartbeat* - signály, kterými se ohlašují u uzlu NameNode, aby se ověřila funkčnost komunikace.

V posledním souboru *mapred-site.xml* nalezneme:

```
<property>
  <name>mapred.job.tracker</name>
  <value>MASTER_NODE:54311</value>
  <description>The host and port that the MapReduce job tracker runs
  at. If "local", then jobs are run in-process as a single map
  and reduce task.
  </description>
</property>

<property>
  <name>mapred.local.dir</name>
  <value>MAPRED_LOCAL_DIR</value>
  <final>true</final>
</property>

<property>
  <name>mapred.tasktracker.map.tasks.maximum</name>
  <value>MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM</value>
  <description>The MH_MAP_TASKS_MAXIMUM will set the maximum amount of
   MAP tasks to be started on a single TASK node, this is either CPU
   or memory bound.</description>
</property>

<property>
  <name>mapred.tasktracker.reduce.tasks.maximum</name>
  <value>MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM</value>
```

```
    <description>The MH_REDUCE_TASKS_MAXIMUM will set the maximum amount
     of  REDUCE tasks to be started on a single TASK node, this is most
      likely   memory bound.</description>
  </property>

  <property>
    <name>mapred.map.tasks</name>
    <value>MAPRED_MAP_TASKS</value>
    <description>The MH_MAP_TASKS is used to hint the application of the
     total  amount of MAP tasks that can be run on the cluster.</description>
  </property>

  <property>
    <name>mapred.reduce.tasks</name>
    <value>MAPRED_REDUCE_TASKS</value>
    <description>The MH_REDUCE_TASKS is used to hint the application of
     the  total  amount of REDUCE tasks that can be run on the cluster.</description>
  </property>
```

Výpis 15: Nastavení myHadoop v souboru mapred-site.xml

Zde je pro nás hlavní hned první nastavení - jedná se o URL adresu, na které poběží JobTracker, uzel, který bude rozdělovat úlohy mezi TaskTracker Slave uzly.

V následujícím kroku si musíme připravit prostředí, resp. proměnné, které nám budou sloužit k navigaci. V souboru *hadoop-1.2.1/conf/hadoop-env.sh* nastavíme cestu k Javě, u které jsme si již dříve ověřili, že ji máme nainstalovanou:

```
export JAVA_HOME=/apps/compilers/java/jdk1.7.0_21
```

Dále musíme změnit cestu na složku, která obsahuje konfiguraci Hadoopu, v základním nastavení se jedná o složku *hadoop-1.2.1/conf*, zde však máme uloženu pouze šablonu. Otevřeme soubor *hadoop-1.2.1/bin/start-all.sh* a změníme cestu k HADOOP_CONF_DIR:

```
export HADOOP_CONF_DIR=~/Workspace/mycluster-conf-$PBS_JOBID
```

Odkázali jsme se na složku, kde si pomocí myHadoopu později vygenerujeme konfiguraci. V proměnné *$PBS_JOBID* je uložena identifikace úlohy, kterou jsme spustili pomocí PBS.

Nyní musíme nastavit proměnné, které využívá myHadoop. Nastavení se provádí v souboru *myhadoop-master/bin/myhadoop-configure.sh*.

```
export JAVA_HOME=/apps/compilers/java/jdk1.7.0_21
export HADOOP_HOME=~/Workspace/hadoop-1.2.1/
export HADOOP_CONF_DIR=~/Workspace/mycluster-conf-$PBS_JOBID
export MH_SCRATCH_DIR=/lscratch/$PBS_JOBID
```

Kromě opětovného nastavení cesty k Javě a k HADOOP_CONF_DIR zde vidíme novou proměnnou, a to MH_SCRATCH_DIR. Tato proměnná uchovává cestu k lokálnímu úložišti uzlu.

Tím jsme dokončili veškerá nastavení a můžeme začít spouštět Hadoop.

Začneme tím, že zažádáme PBS o alokaci zdrojů, v tomto případě zažádáme o 4 uzly:

```
qsub -q qexp -A Hadoop -l select=4:ncpus=16 -l
```

**Poznámka 5.3** Podrobné vysvětlení PBS příkazu nalezneme například v dokumentaci od IT4Inovations[24].

Začneme spuštěním myHadoop skriptu, který nám z šablony vygeneruje nastavení, které následně bude používat Hadoop:

```
[pop0024@cn80 ~]$ Workspace/myhadoop-master/bin/myhadoop-configure.sh
  -p /scratch/pop0024/$PBS_JOBID
myHadoop: Keeping HADOOP_HOME=/home/pop0024/Workspace/hadoop-1.2.1/ from user environment
myHadoop: Keeping MH_SCRATCH_DIR=/lscratch/328076.dm2 from user environment
myHadoop: Keeping HADOOP_CONF_DIR=/home/pop0024/Workspace/mycluster-conf-328076.dm2
  from user environment
myHadoop: Using HADOOP_HOME=/home/pop0024/Workspace/hadoop-1.2.1/
myHadoop: Using MH_SCRATCH_DIR=/lscratch/328076.dm2
myHadoop: Using JAVA_HOME=/apps/compilers/java/jdk1.7.0_21
myHadoop: Generating Hadoop configuration in directory in
  /home/pop0024/Workspace/mycluster-conf-328076.dm2...
myHadoop: Using directory /scratch/pop0024/328076.dm2 for persisting HDFS state...
myHadoop: Designating cn80.bullx as master node (namenode, secondary namenode, and
  jobtracker)
myHadoop: The following nodes will be slaves (datanode, tasktracer):
cn80.bullx
cn79.bullx
cn205.bullx
cn206.bullx
...
```

Parametr *-p /scratch/pop0024/$PBS_JOBID* při spuštění myHadoop kofigurace je informace pro myHadoop, že budeme chtít spustit Hadoop v persistentním módu a v daném umístění uchovávat data i po vypnutí NameNode uzlu. Nejprve nám myHadoop vypsal, jakou cestu použil pro přístup k Hadoopu. Další proměnnou, kterou nastavil, je *MH_SCRATCH_DIR*, v níž je uložená cesta k lokálnímu úložišti každého uzlu. Data na tomto umístění budou vždy po ukončení úlohy vymazána, proto, pokud chceme data zachovat pro další zpracování, používáme persistentní mód. Další proměnnou je *HADOOP_CONF_DIR*, což je cesta ke složce, kterou myHadoop vytvořil. Obsahuje veškerou konfiguraci, pomocí které se následně spustí všechny instance Hadoopu. Obsahuje například také seznam *Master* a *Slave* uzlů. V dalším výpise *Using directory /scratch/pop0024/328076.dm2 for persisting HDFS state...* se můžeme presvědčit, že se Hadoop opravdu bude spouštět v persistentním módu. Dále pak následuje výpis alokovaných uzlů a jejich využití. První přiřazený uzel vždy slouží jako hostující uzel pro Master služby, kterými jsou NameNode, SecondaryNameNode a JobTracker. Následuje výpis uzlů, které budou použity jako Slave uzly pro výpočty a ukládání dat. Můžeme si povšimnout, že mezi uzly je znovu použit uzel, který už hostuje Master služby. To je z toho důvodu, že Master služby jsou poměrně výkonově nenáročné a přiřazený uzel od PBS by tak po většinu času neměl práci a jeho výkon by byl nevyužit.

[24] https://docs.it4i.cz/anselm-cluster-documentation

**Poznámka 5.4** Pokud se nám ve výpise uzlů nevypíšou uzly, ale jsme přesvědčení, že nám je PBS přiřadilo (u Anselmu si to můžeme ověřit přes webové rozhraní), zkontrolujeme následující řádek v souboru *myhadoop-master/bin/myhadoop-configure.sh*:

```
print_nodelist | awk'{print $1}' | sort -u | head -n $NODES > $HADOOP_CONF_DIR/slaves
```

Pro naše účely jej můžeme přepsat na:

```
cat $PBS_NODEFILE > $HADOOP_CONF_DIR/slaves
```

Po vypsání konfigurace následoval výpis:

```
...
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = cn80/10.1.1.80
STARTUP_MSG:   args = [-format, -nonInteractive, -force]
STARTUP_MSG:   version = 1.2.1
...
/lscratch/328076.dm2/namenode_data has been successfully formatted.
15/02/19 09:45:37 INFO namenode.NameNode: SHUTDOWN_MSG:
...
```

V tomto výpise vidíme, že nám myHadoop zároveň připravil složku pro HDFS a rovnou jej také naformátoval (před prvním spuštěním Hadoopu je vždy nutné zformátovat HDFS) a NameNode zase vypnul. Tím jsme zakončili konfigurační část a můžeme začít spouštět Hadoop. Zůstaneme přihlášení na Master uzlu a spustíme Hadoop příkazem:

```
[pop0024@cn80 ~]$ Workspace/hadoop-1.2.1/bin/start-all.sh
starting namenode, logging to ...
cn80.bullx: starting datanode, ...
cn206.bullx: starting datanode, ...
cn205.bullx: starting datanode, ...
cn79.bullx: starting datanode, ...
cn80.bullx: starting secondarynamenode, logging to ...
starting jobtracker, logging to ...
cn206.bullx: starting tasktracker, logging to ...
cn79.bullx: starting tasktracker, logging to ...
cn80.bullx: starting tasktracker, logging to ...
cn205.bullx: starting tasktracker, logging to ...
```

Protože jsme už dříve nastavili cestu ke konfigurační složce, kterou nám vytvořil skript myHadoop, můžeme vidět spuštění jednotlivých služeb na příslušných uzlech. Nakonec si můžeme ověřit správnou funkčnost Hadoopu jednoduchým příkazem:

```
[pop0024@cn80 ~]$ Workspace/hadoop-1.2.1/bin/hadoop dfsadmin -report
Configured Capacity: 1413179392000 (1.29 TB)
Present Capacity: 1341086502957 (1.22 TB)
DFS Remaining: 1341086388224 (1.22 TB)
DFS Used: 114733 (112.04 KB)
DFS Used%: 0%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-------------------------------------------------
Datanodes available: 4 (4 total, 0 dead)
...
```

Dále by následoval výpis jednotlivých DataNode služeb, kolik zbývá prostoru apod. Pro nás je důležitá informace, že NameNode ví o všech svých DataNode neboli že k NameNode došel heartbeat signál od všech DataNode a probíhá správná komunikace mezi uzly. Další možností, jak si ověřit spuštění a chod jednotlivých HDFS uzlů, je přes webové rozhraní, které nalezneme na klasickém portu 50070. V našem případě bychom webové rozhraní našli na adrese *10.1.1.80:50070*, což je adresa uzlu, který aktuálně hostuje NameNode.

Při spuštění myHadoop skriptu jsme zadávali parametr *-p*, jak již bylo řečeno, pomocí tohoto parametru jsme nastavili použití Hadoopu v persistentním módu, tedy ponechání dat na společném úložišti. Data jsou ovšem v HDFS formátu a nelze k nim přistupovat bez spuštěné NameNode služby. Jakmile ukončíme přes PBS aktuální úlohu, na které pracujeme, všechny zdroje se dealokují a při příštím spuštění dostaneme přiřazeny nové zdroje. To znamená, že při dalším spuštění opět musíme začít spuštěním myHadoop skriptu pro vytvoření konfigurační složky. Avšak pokud chceme používat naše již existující data, přes *-p* parametr nastavíme cestu ke složce, kam jsme již dříve ukládali data. Kdybychom nechali proměnnou *$PBS_JOBID*, vytvoří se nám složka nová a HDFS bude prázdný. Pokud se vrátíme k našemu příkladu, použili bychom složku */scratch/-pop0024/328076.dm2*.

Dále s Hadoopem a HDFS můžeme pracovat stejne, jak bylo popsáno v kapitole 3.3.

## 5.2 Apache Mahout

Apache Mahout je open-source knihovna obsahující algoritmy zabývající se strojovým učením[18]. Jak již z názvu vyplývá, je vyvíjen pod záštitou firmy Apache. Zaměřuje se na tři okruhy strojového učení a tzv. kolektivní inteligence a to doporučující algoritmy, shlukování dat, klasifikace. Velkou výhodou Mahoutu je jeho škálovatelnost, můžeme jej použít na data o velikosti kilobytů, ale taky až terabytů. Ke zpracování dat můžeme dokonce použít více strojů zároveň. Mahout je v současné době vydáván jako knihovna tříd napsaných v jazyce Java.

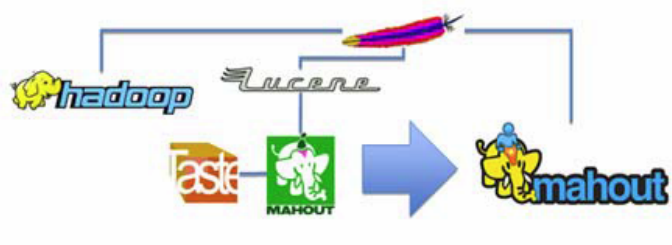
Původně vzniknul jako podprojekt projektu Apache Lucene[18] , což je projekt, který se zabývá vyhledáváním, indexováním v textu apod. V pozdější době se Lucene zaměřilo také na shlukovací algoritmy a část vývojářů se začala věnovat výhradně této větvi, proto se od Lucene nakonec odpojili a založili nový projekt pod názvem Mahout[25].

Původní myšlenkou Mahoutu bylo zaměření se na implementaci všech možných algoritmů pro strojové učení. V dnešní době se zaměřuje pouze na tři hlavní odvětví:

- Doporučující algoritmy (Recommenders)
- Shlukování (Clustering)
- Klasifikace (Clasification)

Dále si je popíšeme podrobněji.

[25]Slovo Mahout pochází z hindštiny a znamená „jezdec na slonovi“, což je odkaz na využívání Hadoopu, který má ve znaku žlutého slona.




Obrázek 18: Rozdělení projektů ve firmě Apache Software Foundation[17]

**Doporučující algoritmy**[18] jsou v dnešní době nejpoužívanější algoritmy z odvětví strojového učení. V komerční sféře je využívají internetové obchody, videopůjčovny, obchody s hudbou a různé seznamky. Například Amazon.com[26] dokonce svým doporučujícím algoritmům věří natolik, že zboží odesílá ještě před tím, než si jej zákazník objedná[27]. Systémy, které využívají doporučující algoritmy, se pokouší zákazníkům doporučit zboží/filmy na základě jeho předešlých akcí - co již nakoupil, na které zboží se dívá nejčastěji apod. Některé systémy také doporučují na základě porovnání zákazníků a doporučují pomocí tzv. k-nejbližšího souseda.

Jak název napovídá, jedná se o **shlukování**[18] dat/informací, které jsou si podobné. Tímto způsobem můžeme třídit a prohledávat velké množství dat. Tímto způsobem můžeme objevit hiearchii a pořádek ve velkých, pro člověka špatně čitelných, datech. Shlukování se dá použít také pro uspořádání velké kolekce tak, aby se dalo lépe vyhledávat, například knihovny mohou shlukovat podle jmen autorů pro snadné vyhledání apod.

**Klasifikační algoritmy**[18] nám pomáhají zjistit, jak je daný soubor součástí nějaké skupiny. Nejznámější využití klasifikačních algoritmů jsou spamfiltry. Často se klasifikační algoritmy učí pravidla rozhodování až za chodu. Rozhodují tedy, zda se nový soubor shoduje s předem zjištěným vzorem, nebo ne.

Aby tyto algoritmy fungovaly správně, potřebují co nejvíce vstupních dat. V některých případech tyto techniky fungují správně i na malých datech, ale stále potřebujeme zajistit výsledek co nejrychleji. Tím se dostáváme k problému škálovatelnosti. Proto je dalším logickým krokem spuštění Mahoutu na Hadoopu, abychom mohli výpočty distribuovat mezi více strojů. V následujících kapitolách si popíšeme jak použít Mahout bez Hadoopu a následně s využitím Hadoopu při využití shlukovacích algoritmů, jejichž implementace bude také popsána.

[26] Jeden z největších internetových obchodů na světě.

[27] urlhttp://www.predictiveanalyticsworld.com/patimes/amazon-knows-what-you-want-before-you-buy-it/

### 5.2.1 Spuštění Apache Mahout

Nejprve si opatříme zdrojový kód Mahoutu. Doporučovaný zdroj jsou oficiální webové stránky Apache Mahout[28]. V našem případě budeme používat verzi 0.9. Stažené soubory rozbalíme a následně můžeme Mahout rovnou zkompilovat pro použití již připravených ukázkových úloh. Jako další krok otevřeme soubor *bin/mahout* a nastavíme cesty:

```
export JAVA_HOME=/usr/lib/jvm/java-7openjdk-amd64
export HADOOP_HOME=~/Workspace/hadoop-1.2.1/
export HADOOP_CONF_DIR=~/Workspace/mycluster-conf-$PBS_JOBID
```

**Poznámka 5.5** Pokud bychom nenastavili proměnnou *HADOOP_HOME* a proměnnou *HADOOP_CONF_DIR*, tak by se Mahout spouštěl v tzv. „stand-alone“ módu, dá se tedy spustit i bez použití Hadoopu.

Dalším krokem je spuštění Hadoop služeb. Jakmile máme Hadoop spuštěn, překopírujeme na HDFS následující ukázkový vstupní soubor *affinity.txt*:

```
0,0,0
0,1,0.8
0,2,0.5
1,0,0.8
1,1,0
1,2,0.9
2,0,0.5
2,1,0.9
2,2,0
```

```
[pop0024@cn80 ~]$ Workspace/hadoop-1.2.1/bin/hadoop fs -mkdir input
[pop0024@cn80 ~]$ Workspace/hadoop-1.2.1/bin/hadoop fs -put
~/Workspace/Data/affinity.txt input/
```

V dalším kroku již můžeme spustit ukázkovou úlohu na SpectralClustering, kterou Mahout obsahuje již impementovanou v základním balíčku, spustíme ji pomocí příkazu:

```
[pop0024@cn80 ~]$ Workspace/mahout-comp/bin/mahout spectralkmeans
  -i input/
  -o output/
  -d 3
  -k 2
  -x 10
```

Kde jednotlivé proměnné znamenají:

- **-i** – jméno vstupního souboru, v našem případě předem připravená *input* složka na HDFS
- **-o** – jméno složky, která bude sloužit pro výstupní soubory
- **-d** – počet sloupců matice
- **-k** – počet výsledných shluků a zároveň počet vlastních vektorů
- **-x** – maximální počet k-means iterací

[28] http://svn.apache.org/repos/asf/mahout/trunk

Tímto příkazem se nám do naší *output* složky vytvořily výstupní shluky. Abychom je mohli přečíst, použijeme další nástroj Mahoutu a to *clusterdump*:

```
[pop0024@cn80 ~]$ Workspace/mahout-comp/bin/mahout clusterdump -i
output/kmeans_out/clusters-1-final/ -p output/kmeans_out/clusteredPoints/ -of TEXT
```

Kde:

- **-i** – zastupuje vstupní soubor, v našem případě jako vstup slouží výstup z předešlé úlohy
- **-p** – odkaz na složku obsahující jednotlivé shlukované body
- **-of** – určuje formát výstupu, v našem případě je zvolen jednoduchý text

Můžeme si povšimnout, že například chybí parametr *-o*. Tento parametr představuje název výstupního souboru a je nepovinný, pokud jej nezadáme, výsledek se pouze vypíše do konzole. Seznam všech možných parametrů nalezneme na stránkách Apache Mahout[29]. Po spuštění předchozího příkazu dostaneme následující výpis:

```
...
VL-0{n=3 c=[0.412, 0.448] r=[0.766, 0.208]}
Weight : [props - optional]:  Point:
1.0 : [distance=1.2591040468689667]: 0 = [-0.671, 0.742]
1.0 : [distance=0.3147760117172418]: 2 = [0.954, 0.301]
VL-1{n=2 c=[-0.288, -0.958] r=[]}
Weight : [props - optional]:  Point:
1.0 : [distance=0.0]: 1 = [-0.288, -0.958]
...
```

### 5.2.2 Apache Mahout v HPC prostředí

Mahout máme již od začátku připraven pro spuštění v HPC prostředí. Důkazem může být například to, že příklady v předešlé kapitole byly na HPC spouštěny. Ve výpisu příkazů můžeme vidět označení *cn80*, což je označení uzlu, který nám byl přidělen.

Abychom využili plný výkon superpočítače, musíme nastavit příslušný počet mapovacích a redukčních úloh, které se mohou spouštět na jednom uzlu. Toto nastavení velice ovlivňuje výsledné využití zdrojů. Ve výpise 15 můžeme najít příslušná nastavení. Pokud nastavení nebudeme měnit, dosadí se základní hodnoty, které povolují spuštění maximálně dvou úloh na uzlu, což je pro uzel s 16 procesory velice málo. Ve výsledku by se pak používalo pouze zhruba 6% výkonu. Konkrétně budeme dosazovat za tyto hodnoty:

- MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM - nastavíme 16, tato hodnota nastaví maximum mapovacích úloh, které mohou být spuštěny na jednom TaskNode
- MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM - nastavíme 16, tato hodnota nastaví maximum redukčních úloh, které mohou být spuštěny na jednom TaskNode

[29] https://mahout.apache.org/users/clustering/clusteringyourdata.html

- MAPRED_MAP_TASKS - nastavíme 64, tato hodnota nastaví maximum mapovacích úloh pro cluster
- MAPRED_REDUCE_TASKS - nastavíme 64, tato hodnota nastaví maximum redukčních úloh pro cluster

Popis jednotlivých položek vidíme taktéž ve výpise 15. Soubor, který takto chceme upravit, je *hadoop-1.2.1/conf/mapred-site.xml*. Jedná se o šablonu, která se pak používá pro vygenerování nastavení pomocí myHadoop. Nastavení, pomocí kterých můžeme ovlivnit výkon Hadoopu, je více, například můžeme nastavit velikost alokované paměti pro jednu úlohu apod. Z testů, které jsou podrobněji popsány v kapitole 5.3, vyplývá, že s tímto nastavením vytížíme při složitých výpočtech uzel Anselmu téměř na 100%, konkrétně se pohybujeme kolem hodnot 99,6%. Vytížení však není stálé a velice kolísá, ve výsledku použijeme pouze 60-70% přiděleného výpočetního výkonu. Je tedy zřejmé, že pokud budeme chtít využít více výkonu, musíme tyto údaje nastavit lépe. Podrobnější měření a grafy si ukážeme u jednotlivých spuštěných úloh v kapitole 5.3.

Poslední parametr, který potřebujeme nastavit pro použití Mahoutu ke shlukvování velkých dat, je maximální hodnota paměti, kterou můžeme použít. V základním nastavení Mahout má nastavenou jako maximální hodnotu 1 000MB. Pro práci s velkými soubory potřebujeme tuto hodnotu zvýšit - docílíme toho úpravou souboru *bin/mahout*, k definování proměnných dopíšeme následující:

```
export MAHOUT_HEAPSIZE=10000
```

Jak výpis napovídá, jedná se o zvýšení maximální hodnoty, kterou může Mahout použít, nastavíme 10 000MB.

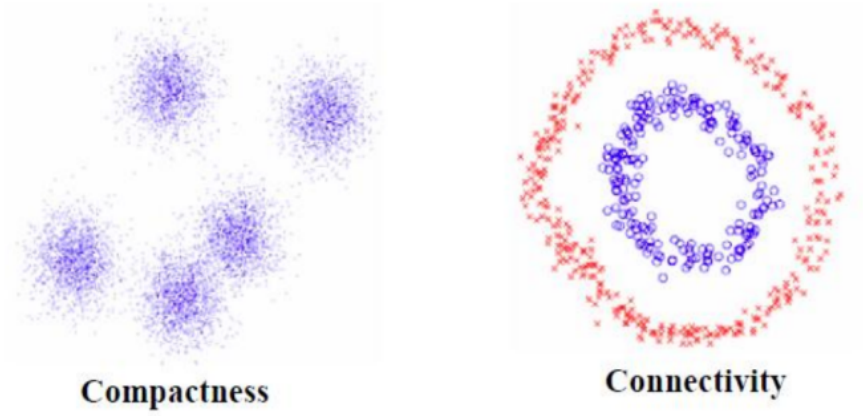
#### 5.2.2.1 Spektrální shlukování

Algoritmus, který zvolíme pro otestování Hadoopu v HPC, je shlukovací algoritmus, konrétně se jedná o spektrální shlukování[8][20].

Spektrální shlukování, jak již jeho název napovídá, využívá spektra (nebo eigenvalues) z matice podobnosti. Narozdíl od jiných shlukovacích algoritmů (například k-means), které zkoumají hustotu (*compactness*) dat, algoritmus spektrálního shlukování zkoumá propojení (*connectivity*) dat. Rozdíl v těchto dvou přístupech můžeme vidět na obrázku 19. Jsou tedy situace, kdy je vhodnější použít k-means a kdy zase spektralní shlukování, například při segmentaci obrazu.

Apache Mahout provádí spektrální shlukování v těchto čtyřech základních krocích[19]:

- Výpočet matice podobnosti z dostupných dat. To zahrnuje také volbu podobnostní funkce.
- Výpočet Laplaceova grafu z matice podobnosti. Existuje více druhů Laplaceových grafů; který graf se použije často záleží na situaci.
- Výpočet eigenvectors a eigenvalues z Laplaceovy matice. Tento výpočet ovlivňuje parametr -k, který se zadává při spuštení spektralního shlukování v Apache Mahout.




Obrázek 19: Porovnání hustoty (Compactness) a propojení (Connectivity) dat[8]

- Použití eigenvectors pro shlukovací algorimus k-means.

Jako vstupní data pro Mahout slouží matice podobnosti v textovém souboru. V tomto souboru každý řádek zastupuje jeden element matice. Jednotlivé řádky se skládají z trojic:

```
i, j, value
```

Kde *i* a *j* jsou indexy v matici a value pak znázorňuje vztah mezi těmito prvky.

## 5.3 Testování výkonu Hadoopu v HPC

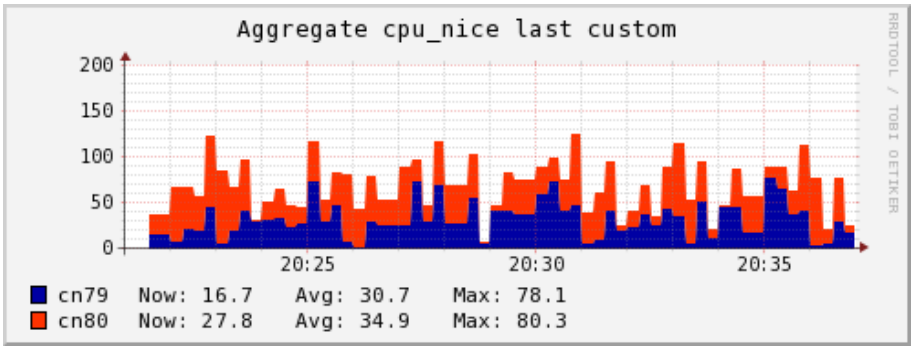
Pro testování zvolíme dvě sady dat. Jedna data budou malá a budou obsahovat pouze pár záznamů, na těchto datech se pokusíme odhadnout, jaká nastavení lze následně využít při testování na Big Data, teda na velkých datech.

### 5.3.1 Malá data

Jak již bylo řečeno dříve, testovat budeme pomocí shlukovacího algoritmu. Jako první vstupní data použijeme řídkou matici podobnosti, která má:

- 1 760 řádků
- 85 unikátních bodů
- velikost souboru: 39KB

Tento soubor neodpovídá definici *big data*. Použijeme jej pro otestování funkčnosti algoritmu. Jako spouštěcí skript pro úlohu použijeme skript ve výpise 20, který nalezneme v příloze.




Obrázek 20: Test výkonu na 2x16

Parametry, kterými můžeme nejvíce ovlivnit rychlost výpočtu, je počet použitých jader a HPC clusterů a následné nastavení čtyř hodnot, které řídí rozložení práce v Hadoopu, jak bylo popsáno v kapitole 5.2.2.

K dalšímu testování využijeme výše popsaná data a skript pro spuštění úlohy (výpis 20).

### Cluster 2x16 jader

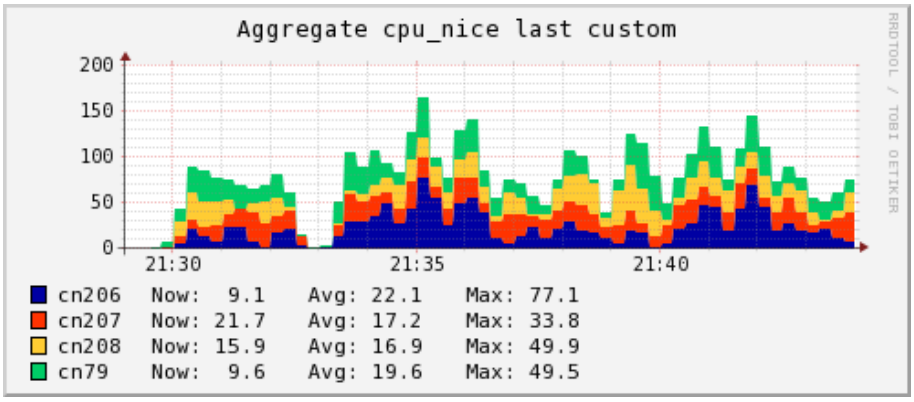
Jako první vyzkoušíme dva uzly, kde každý má 16 jader. Ostatní parametry nastavíme následovně:

- MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM = 32
- MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM = 32
- MAPRED_MAP_TASKS = 32
- MAPRED_REDUCE_TASKS = 32

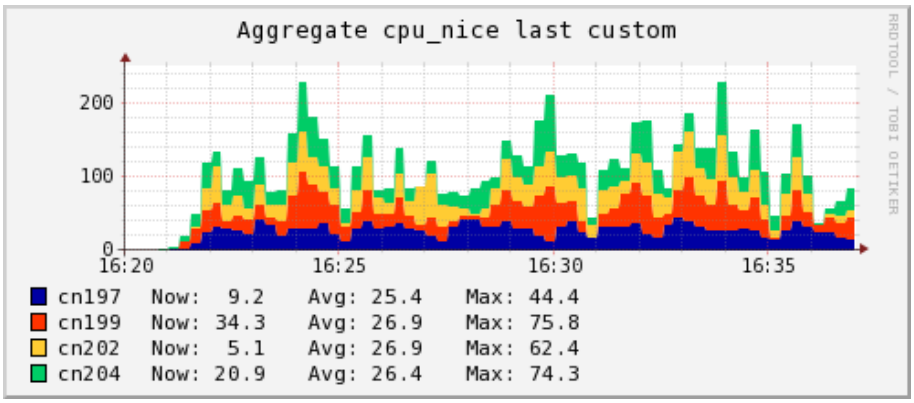
Z nastavení vyplývá, že by každé jádro mělo obsluhovat jednu mapovací a jednu redukční úlohu. Procesory by tedy měly být teoreticky dostatečně zatíženy. Výsledné zatížení můžeme vidět na grafu na obrázku 20. Vidíme, že procesory nebyly zatíženy tak, jak bychom potřebovali. Průměrné zatížení kolem 30% je nedostatečné. Tomu odpovídá i výsledný čas shlukování, který pro tato malá data byl **17,1009 minut**.

### Cluster 4x16 jader

Dalším krokem bude vyzkoušet zvýšit počet clusterů neboli výsledný počet jader. Ostatní nastavení ponecháme stejná jako v předchozím testu. V tomto testu tedy použijeme 4 uzly, kdy každý má 16 jader. Teoreticky bychom měli dosáhnout mnohem lepšího výsledku jako v předchozím případě. Využití přidělených prostředků můžeme vidět na grafu na obrázku 21. Opět vidíme velké nevyužití výkonu. Průměrná hodnota zátěže je tentokrát ještě menší a to kolem 20% na uzel. Výsledný čas shlukování se díky tomu zlepšil pouze nepatrně - **15,1781 minut**.




Obrázek 21: Test výkonu na 4x16



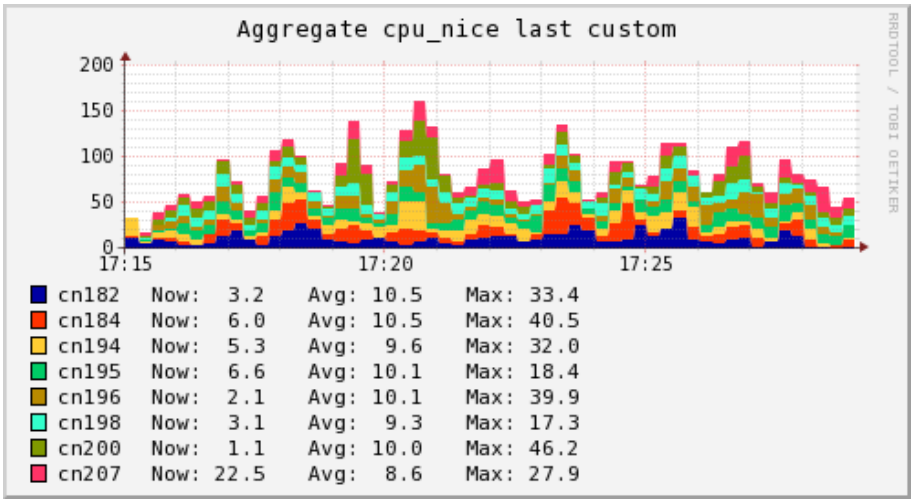

Obrázek 22: Test výkonu na optimalizovaném 4x16

### Optimalizace pro cluster 4x16 jader

Můžeme se pokusit vylepšit předchozí výsledek tím, že zvýšíme počet úloh na cluster. Pokud zvýšíme počet úloh na jednotlivé TaskNode, stává se Hadoop nestabilní a setkáme se s častými pády. Parametry tedy změníme následovně:

- MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM = 32
- MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM = 32
- MAPRED_MAP_TASKS = 48
- MAPRED_REDUCE_TASKS = 48

Výsledek tohoto pokusu vidíme na obrázku 22. Můžeme vidět nepatrné zlepšení v průměrném využití výkonu. Zhruba na 25%. Tomu však neodpovídá výsledný čas, který se naopak zvýšil na **16,17 minut**.




Obrázek 23: Test výkonu na 8x16

#### Cluster 8x16 jader

Nejlépe se nám osvědčilo nastavení z druhého pokusu, necháme tedy všechny parametry nastaveny na 32. Zároveň zkusíme zvýšit počet uzlů na dvojnásobek, abychom zjistili, zda se zkrátí doba trvání výpočtu. Výsledek si můžeme prohlédnout na obrázku 23. Průměrné využití kolem 10% je opravdu málo. Výsledný čas se zlepšil na **14,2666 minut**, ale za cenu nevyužitého výkonu. Je tedy jasné, že bychom potřebovali lépe nastavit rozdělování úloh.

### 5.3.2 Testování Big Data

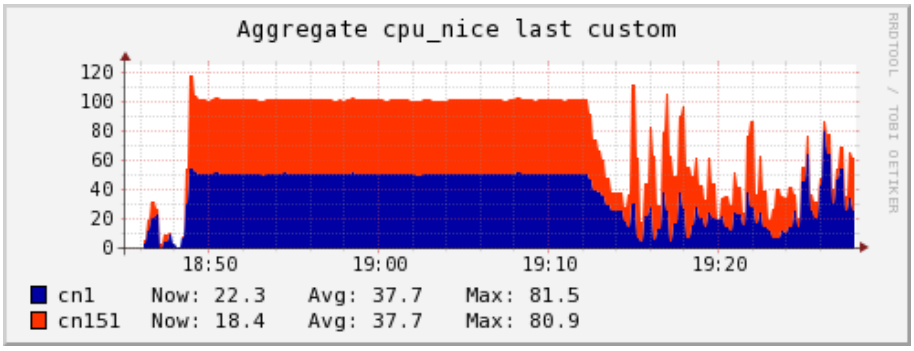
Důvodem, proč se nám nepodařilo využít přidělený výkon v předchozích případech, může být malá velikost dat. Celková úloha se rozloží na hodně malých, které se vykonávají rychle a stále se spouští nové, nedaří se tedy práci dostatečně distribuovat. Proto vyzkoušíme stejný pokus s následujícími daty, kde by měly být jednotlivé výpočty složitější:

- 7 818 830 řádků
- 1 098 917 unikátních bodů
- velikost souboru: 268 817KB

Tento soubor už více odpovídá definici BigData. Nad těmito daty opět spustíme stejný algoritmus spektrálního shlukování.

#### Cluster 2x16 jader

Začneme stejným testem jako v případě malých dat. První úlohu spustíme na clusteru 2x16 jader a nastavíme Hadoop následovně:




Obrázek 24: Test výkonu na 2x16 - BigData

- MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM = 16
- MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM = 16
- MAPRED_MAP_TASKS = 16
- MAPRED_REDUCE_TASKS = 16

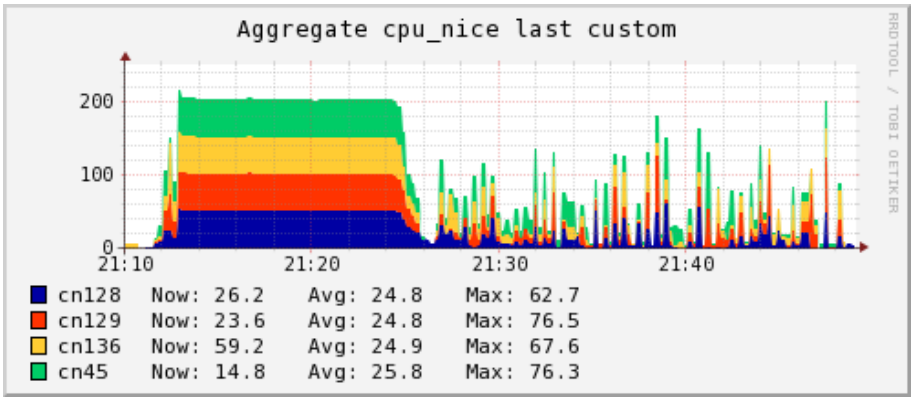
Oproti testu na malých datech jsme pro začátek snížili maximální počet úloh spuštěných na clusteru. To proto, že s většími daty jsou větší nároky na pamět a při velkém množství naráz spuštěných úloh Mahout padá. Toto nastavení však zvládne bez problémů a na Obrázku 24 vidíme výsledky z měření. Na grafu si můžeme povšimnout, že při použití větších dat se lépe distribuuje práce mezi uzly. Vytížení je rovnoměrné, avšak jakmile se dostaneme do druhé části algoritmu, kde se počítají jednotlivé k-means iterace, distribuce se zhorší a využití výkonu opět klesá. To je bohužel dáno vlastností algoritmu a horší distribucí této zátěže. Výsledné průměrné využití na jeden uzel bylo 37% a test trval **43,7 minut**.

### Cluster 4x16 jader

Na větším clusteru opět nastavíme parametry:

- MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM = 32
- MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM = 32
- MAPRED_MAP_TASKS = 32
- MAPRED_REDUCE_TASKS = 32

Samozřejmě, že bychom mohli vyzkoušet jiný maximální počet mapovacích úloh a jiný maximální počet redukčních úloh, ale pro základní otestování ponecháme hodnoty stejné. Na Obrázku 25 vidíme výsledek měření, kde průměrné využití na uzel bylo 25%. V druhé části algoritmu se zvýraznil již zmíněný problém s počítáním k-means iterací.




Obrázek 25: Test výkonu na 4x16 - BigData

Výsledný čas se změnil na rovných **39 minut**, což při zdvojnásobení dostupných zdrojů a povolení vyššího počtu maximálních spuštěných úloh není moc velké zlepšení.

### Cluster 8x16 jader

Ponecháme stejné nastavení jako v předchozím případě:

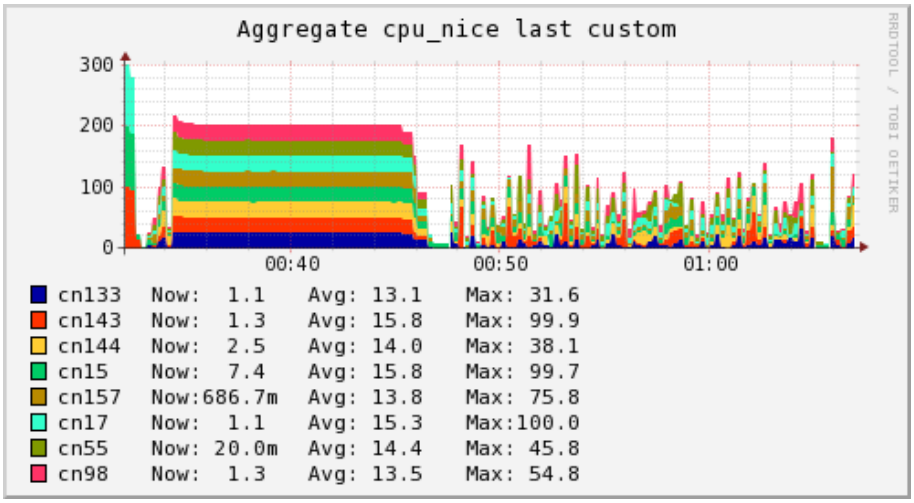
- MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM = 32
- MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM = 32
- MAPRED_MAP_TASKS = 32
- MAPRED_REDUCE_TASKS = 32

Z předcházejících testů se dá se předpokládat, že při pouhém zvětšení clusteru, ale nevyladění parametrů se opět sníží průměrné využití na cluster a čas výpočtu. Výsledek tohoto testu vidíme na Obrázku 26 a naše předpoklady se potvrzují. Průměrné využití výkonu na uzel je kolem 15% a výsledný čas výpočtu byl **34,5 minut**.

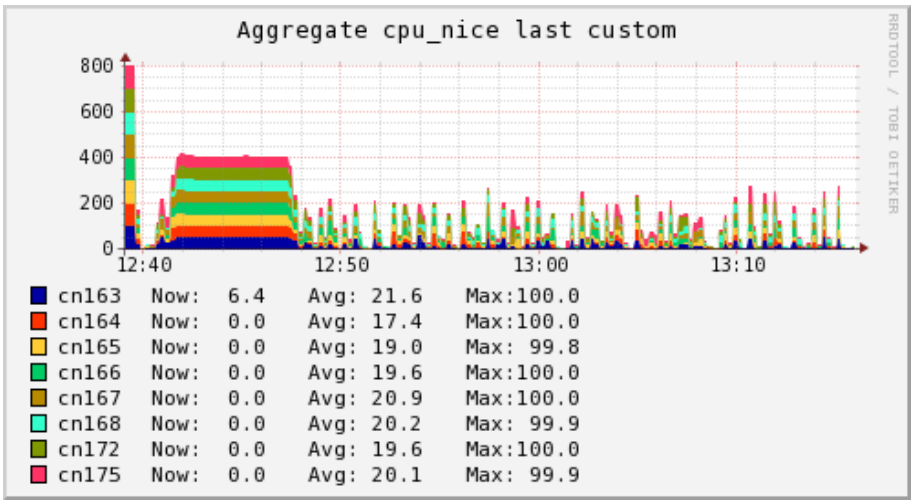
Zvětšováním clusteru se nám daří snižovat výsledný čas výpočtu, ale za cenu plýtvání výkonem. Na tomto clusteru zkusíme vylepšit výkon úpravou parametrů:

- MAPRED_TASKTRACKER_MAP_TASKS_MAXIMUM = 32
- MAPRED_TASKTRACKER_REDUCE_TASKS_MAXIMUM = 32
- MAPRED_MAP_TASKS = 64
- MAPRED_REDUCE_TASKS = 64

Maximální hodnoty na jeden uzel ponecháme, ale zvýšíme celkové maxima na cluster. Výsledek tohoto měření je na Obrázku 27. Vidíme, že průměrné využití na uzel se zvýšilo na 19-20%, ale zhoršila se druhá část, kde se počítají k-means iterace, takže výsledný čas výpočtu se zvýšil na **36,35 minut**.




Obrázek 26: Test výkonu na 8x16 - BigData




Obrázek 27: Test výkonu na 8x16 s vylepšenými parametry - BigData

### 5.3.3 Shrnutí měření

Z předešlých testů vyplývá, že zvětšování počtu uzlů v clusteru výpočty zrychluje, ale bez správně nastevených parametrů se hodně plýtvá výkonem. Pro nejlepší využití výkonu je potřeba najít správný poměr mapovacích a redukčních úloh pro daný problém, případně změnit velikosti jednotlivých bloků tak, aby Mahout neměl problémy s pamětí a dokázal pracovat i s větším počtem úloh. Pokud si shrneme výsledky předchozích měření, tak nejlépe vychází měření na nejmenším clusteru (2x16), kde využití bylo kolem 37% na uzel a čas výpočtu byl 43 minut. Při ostatních nastaveních se hodně plýtvalo výkonem, ale jak již bylo řečeno, tento problem by se mohl dát řešit lépe nastavenými parametry.

### 5.3.4 Vytvoření shluků - ClusterDump

V předchozí kapitole jsme pomocí Mahoutu provedli nad daty algoritmus spektrálního shlukování. Nyní musíme data z HDFS dostat v čitelné podobě. Na tento problém použijeme nástroj *ClusterDump*, který je součástí balíčku Apache Mahout. Pomocí tohoto nástroje si můžeme zvolit typ výstupního souboru. V základní nabídce jsou přichystány formáty Text, CSV, JSON nebo GRAPH_ML. V kapitole 5.2.1 jsme mohli vidět výsledek vypsaný v obyčejné textové podobě.

Můžeme si však snadno dopsat vlastní formát výstupu. Toho docílíme úpravou sobouru *ClusterDumper.java*. V tomto souboru vyhledáme výpis výčtového typu a dopíšeme si vlastní. V našem případě je nazván nový formát jako MF - MyFormat, ale název je samozřejmě libovolný.

```
public enum OUTPUT_FORMAT {
    TEXT,
    CSV,
    MF,
    GRAPH_ML,
    JSON,
 }
```

Výpis 16: Výčtový typ pro výstupní formáty nástroje ClusterDump

Novou volbu také přidáme k parametru „/of“ pomocí úpravy tohoto řádku:

```
addOption(OUTPUT_FORMAT_OPT, "of", "The␣optional␣output␣format␣for␣the␣results.␣␣␣
      Options:␣TEXT,␣CSV,␣MF,␣JSON␣or␣GRAPH_ML",
        "TEXT");
```

Výpis 17: Úprava parametru v nástroji ClusterDump

Dále vyhledáme v této třídě následující blok kódu, kde se rozhoduje, jaký výstupní formát se použije. Přidáme další možnost a opět použijeme pro náš nový formát označení MF:

```
        case TEXT:
          result = new ClusterDumperWriter(writer, clusterIdToPoints, measure, numTopFeatures,
              dictionary, subString);
```

```
        break;
      case CSV:
        result = new CSVClusterWriter(writer, clusterIdToPoints, measure);
        break;
      case GRAPH_ML:
        result = new GraphMLClusterWriter(writer, clusterIdToPoints, measure, numTopFeatures,
            dictionary, subString);
        break;
      case JSON:
        result = new JsonClusterWriter(writer, clusterIdToPoints, measure, numTopFeatures,
            dictionary);
        break;
      case MF:
          result = new MFClusterWriter(writer, clusterIdToPoints, measure);
          break;
      default:
        throw new IllegalStateException("Unknown outputformat: " + outputFormat);
    }
```

Výpis 18: Výběr výstupního formátu v nástroji ClusterDump

Ve výpise 18 můžeme vidět, že se odkazujeme na novou třídu, která se v daném souboru již nenachází. V této třídě se bude nacházet samotná logika a formát výstupního typu. Tuto třídu si vytvoříme a naimplementujeme tak, aby vracela formát v následujícím formátu:

```
0_4 3 LEAF:103,553,660
```

Kde:

- 0_4 - označení shluku
- 3 - celkový počet objektů ve shluku
- LEAF: - klíčové slovo, které označuje začátek výpisu objektů ve shluku

Třídu, která tento výstupní formát implementuje, nalezneme v příloze ve výpise 21. Nakonec Mahout zkompilujeme a nový formát můžeme dále používat pomocí parametru „-of MF“. Pokud si necháme vypsat výsledek z testovací úlohy, kterou jsme používali v kapitole 5.3, dostaneme následující výpis:

```
0_0 22 LEAF:0,3,9,19,26,27,33,35,46,49,51,52,55,56,57,61,62,65,69,72,82,84
0_1 1 LEAF:11
0_2 4 LEAF:6,10,38,53
0_3 8 LEAF:14,30,42,44,48,66,67,83
0_4 12 LEAF:7,12,13,15,20,24,28,36,37,40,45,59
0_5 3 LEAF:5,34,39
0_6 16 LEAF:16,18,25,47,54,63,64,68,70,73,74,75,77,78,79,81
0_7 7 LEAF:4,22,29,32,41,71,76
0_8 10 LEAF:1,2,8,17,21,31,43,50,60,80
0_9 2 LEAF:23,58
```

# 6 Závěr

V této diplomové práci byla popsána technologie Hadoop a následně jsme si ověřili funkčnost vybraných frameworků z Hadoop ekosystému. Prvním testovaným byl framework Oozie, který poskytuje nové možnosti pro plánování úloh. V praxi se Oozie používá společně s frameworkem HUE, který použití zjednodušuje a zpříjemňuje. Dalším testovaným frameworkem byl Hive. Tento framework nám pomáhá spravovat data na HDFS a přibližuje tak práci s daty blíže ke známějším relačním databázím. V praxi se opět používá s již zmiňovaným HUE frameworkem, který celý Hadoop ekosystém zastřešuje a poskytuje tak přístup ke všem možnostem Hadoopu z jednoho místa.

Hlavním cílem této práce bylo nasazení technologie Hadoop v HPC prostředí, což se také podařilo. Z výsledků následného testování vyplývá, že je reálné používat Hadoop v HPC prostředí, i když pro něj nebyl původně navržen.

Narazil jsem však na problém škálovatelnosti. Zdá se, že pouhé zvyšování počtu uzlů v clusteru nestačí a musíme se zaměřit na nastavení Hadoopu z hlediska počtu mapovacích a redukčních úloh na jednotlivé procesory. Nevadí-li nám plýtvání výkonem, který nám poskytují přidělené zdroje, můžeme neustále zvyšovat počet uzlů v clusteru a výpočty budou v Hadoopu trvat kratší dobu. Nejlépe vychází měření na nejmenším clusteru (2x16), kde využití přidělených zdrojů bylo kolem 37 % na uzel a čas výpočtu byl 43 minut.

Je potřeba si uvědomit, že Hadoop je určen pro použití na běžně dostupných počítačích a měl by pracovat pouze s procesory o dvou až čtyřech jádrech a v HPC prostředí jej spouštíme na procesorech s 16 jádry. V další fázi práce je proto potřeba zaměřit se na detailnější nastavení konfiguračních parametrů Hadoopu tak, aby zvládal práci se zdroji, jaké mu HPC nabízí.

# 7 Reference


[1] Tom White, *Hadoop: The Definitive Guide, 3rd Edition,* O'Reilly Media / Yahoo Press, 2012.

[2] Graeme Noseworthy, IBM, *The Flood of Big Data* [online] 2012 [cit. 2014-11-20]. Dostupné z: `http://analyzingmedia.com/2012/infographic-big-flood-of-big-data-in-digital-marketing/`

[3] Lam Chuck, *Hadoop in Action,* Manning Publications, 2011.

[4] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler *The Hadoop Distributed File System,* Yahoo! Sunnyvale, California USA, 2010.

[5] Jeffrey Dean, Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*, COMMUNICATIONS OF THE ACM, 2008.

[6] Sriram Krishnan, Mahidhar Tatineni, Chaitanya Baru, *myHadoop - Hadoop-on-Demand on Traditional HPC Resources,* San Diego Supercomputer Center, 2011.

[7] Sriram Krishnan, *myHadoop 0.2a: Hadoop-on-demand on Traditional HPCResources*, San Diego Supercomputer Center, University of California at San Diego.

[8] Charles H Martin, *Spectral Clustering: A quick overview* [online] 2012 [cit. 2015-03-15]. Dostupné z: `https://charlesmartin14.wordpress.com/2012/10/09/spectral-clustering/`

[9] Michal Marinič, *Škálovatelné predzpracování dat prostrednictvím nástroje Hadoop*, Vysoké učení technické v Brně.

[10] Philip Russom, *Big Data Analytics,* TDWI RESEARCH, 2011.

[11] Sandeep Raut, *Hadoop Simplified* [online] 2013 [cit. 2015-01-20]. Dostupné z: `http://simplified-analytics.blogspot.cz/2013/08/hadoop-simplified.html`

[12] Apache Software Foundation, *Oozie Specification, a Hadoop Workflow System* [online] 2012 [cit. 2014-11-22]. Dostupné z: `https://oozie.apache.org/docs/3.1.3-incubating/WorkflowFunctionalSpec.html`

[13] Apache Software Foundation, *Oozie Specification, a Hadoop Workflow System* [online] 2015 [cit. 2015-03-15]. Dostupné z: `https://cwiki.apache.org/confluence/display/Hive/Home`

[14] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff and Raghotham Murthy, *Hive - A Warehousing Solution Over a Map-Reduce Framework,* Facebook Data Infrastructure Team.

[15] Cloudera, *Hue 2 User Guide,* Cloudera Inc. , 2013.

[16] Cloudera, *Hue Configuration* [online] 2015 [cit. 2015-04-05]. Dostupné z: http://www.cloudera.com/content/cloudera/en/documentation/cdh4/v4-2-2/CDH4-Installation-Guide/cdh4ig_topic_15_6.html

[17] Thanachart Numnonda, *Big Data Analytics Using Mahout* [online] 2015 [cit. 2015-04-10]. Dostupné z: http://www.slideshare.net/imcinstitute/big-data-analytics-using-mahout

[18] Sean Owen, Robin Anil, Ted Dunning, Ellen Friedman, *Mahout In Action,* M A N N I N G, 2011.

[19] Apache Software Foundation, *Spectral Clustering Overview* [online] 2014 [cit. 2015-05-03]. Dostupné z: https://mahout.apache.org/users/clustering/spectral-clustering.html

[20] Jiří Kléma, *Shluková analýza – specializované algoritmy,* Katedra kybernetiky FEL, ČVUT Praha.

# A Ukázková MapReduce úloha Wordcount

```
package org.apache.hadoop.examples;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

```
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String [] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Výpis 19: Implementace ukázkové úlohy Wordcount

# B Skripty pro spouštění úloh

```
#!/bin/bash

#myHadoop configuration
export MYHADOOP_HOME=~/Workspace/myhadoop-master
export HDFS_DIR=/scratch/pop0024/small_$PBS_JOBID

#Hadoop configuration
export HADOOP_HOME=~/Workspace/hadoop-1.2.1
export LOCAL_INPUT_DIR=~/Downloads/Data/com_output.csv
export LOCAL_OUTPUT_DIR=~/Downloads/Data/outputs/small_$BS_JOBID
export HDFS_INPUT_DIR=input
export HDFS_OUTPUT_DIR=spectoutput

#Mahout configuration
export MAHOUT_HOME=~/Workspace/mahout-comp

#SpectralKmeans configuration
export DIMENSION=84
export CLUSTERS=10
export KMEANS_ITER=5

date

#myHadoop
$MYHADOOP_HOME/bin/myhadoop-configure.sh -p $HDFS_DIR

#Hadoop
$HADOOP_HOME/bin/start-all.sh
$HADOOP_HOME/bin/hadoop fs -mkdir $HDFS_INPUT_DIR
$HADOOP_HOME/bin/hadoop fs -put $LOCAL_INPUT_DIR $HDFS_INPUT_DIR/

#Mahout
$MAHOUT_HOME/bin/mahout spectralkmeans -i $HDFS_INPUT_DIR -o $HDFS_OUTPUT_DIR
    -d $DIMENSION -k $CLUSTERS -x $KMEANS_ITER

#Get results from HDFS to local
$HADOOP_HOME/bin/hadoop fs -get $HDFS_OUTPUT_DIR $LOCAL_OUTPUT_DIR

#Dump clusters from HDFS to local
$MAHOUT_HOME/bin/mahout clusterdump -i $HDFS_OUTPUT_DIR/kmeans_out/clusters-1-
    final/ -o
clusterdump_10.csv -of CSV -p $HDFS_OUTPUT_DIR/kmeans_out/clusteredPoints/

#Delete temp config files
rm -r ~/Workspace/mycluster-conf-$PBS_JOBID

date
```

Výpis 20: Šablona pro skript spouštějící úlohu v HPC prostředí

# C Nový formát pro nástroj ClusterDump

```
package org.apache.mahout.utils.clustering;

import org.apache.mahout.clustering.Cluster;
import org.apache.mahout.clustering.classify.WeightedPropertyVectorWritable;
import org.apache.mahout.clustering.iterator.ClusterWritable;
import org.apache.mahout.common.distance.DistanceMeasure;
import org.apache.mahout.math.NamedVector;
import org.apache.mahout.math.Vector;

import java.io.IOException;
import java.io.Writer;
import java.util.List;
import java.util.Map;
import java.util.regex.Pattern;

public class MFClusterWriter extends AbstractClusterWriter {

  private static final Pattern VEC_PATTERN = Pattern.compile(" \\{|\\:|\\,|\\} ");

  public MFClusterWriter(Writer writer, Map<Integer, List<WeightedPropertyVectorWritable>>
      clusterIdToPoints,
      DistanceMeasure measure) {
    super(writer, clusterIdToPoints, measure); }

  @Override
  public void write(ClusterWritable clusterWritable) throws IOException {
    StringBuilder line = new StringBuilder();
    Cluster cluster = clusterWritable.getValue();
    line.append("0_" + cluster.getId());
    List<WeightedPropertyVectorWritable> points = getClusterIdToPoints().get(cluster.getId());
    if (points != null) {
      line.append("␣"+points.size()+"␣");
      line.append("LEAF:");
      for (WeightedPropertyVectorWritable point : points) {
        Vector theVec = point.getVector();
        if (theVec instanceof NamedVector) {
          line.append(((NamedVector)theVec).getName());
        } else {
          String vecStr = theVec.asFormatString();
          vecStr = VEC_PATTERN.matcher(vecStr).replaceAll("_");
          line.append(vecStr);
        }
        line.append(',');
      }
      line.deleteCharAt(line.length() - 1);
      getWriter().append(line).append("\n");
    }
  }
}
```

Výpis 21: Implementace nového výstupního formátu

# D DVD-ROM

Obsah přiloženého DVD:

- Balíček Hadoop 1.2.1
- Balíček Mahout 0.9
- Balíček myhadoop-master.zip
- Script myHadoop.sh, který spouští testování - nutná úprava parametrů, zadávající cesty k souborům
- Dataset Bookset.zip pro ukázkovou úlohu s Hive
- Třída Wordcount.java obsahující implementaci ukázkové úlohy Wordcount
- Testovací data v souboru *dblp.gdf.csv*